Software Composition with Multiple Nested Inheritance

Nathaniel Nystrom Xin Qi Andrew C. Myers

Computer Science Department
Cornell University
{nystrom,qixin,andru}@cs.cornell.edu

Abstract

This paper introduces a programming language that makes it convenient to extend large software systems and even to compose them in a modular way. JX/MI supports *multiple nested inheritance*, building on earlier work on nested inheritance in the language JX. Nested inheritance permits modular, type-safe extension of a package (including nested packages and classes), while preserving existing type relationships. Multiple nested inheritance enables simultaneous extension of *two or more* classes or packages, composing their types and behavior while resolving conflicts with a relatively small amount of code. The utility of JX/MI is demonstrated by using it to construct two composable, extensible frameworks: a compiler framework for Java, and a peer-to-peer networking system. Both frameworks support composition of extensions. For example, two compilers adding different, domain-specific features to Java can be composed to obtain a compiler for a language that supports both sets of features.

1. Introduction

Software is frequently produced by combining and extending preexisting components; it is therefore valuable for programming languages to support code reuse, extension, and composition. Existing mechanisms like class inheritance and functors address the problem of code reuse and extension for small or simple extensions, but do not work well for larger bodies of code such as compilers or operating systems, which are composed of many mutually dependent classes, functions, and types. Better language support is needed to build these larger systems in an extensible way.

For example, a body of software written in an object-oriented language usually comprises many classes related through inheritance. Ordinary class inheritance can extend the software by adding new classes to the leaves of the class hierarchy, but in general, more significant changes may be needed to construct the extended software. For instance, all of the classes in some part of the class hierarchy may need (the same) additional field. Ordinary object-oriented languages provide no means to make this kind of extension in a modular fashion. The non-modular solution of copying and modifying the code is not acceptable because it duplicates code and makes maintenance of the copied code difficult as the original code evolves.

This paper introduces the language JX/MI, which supports the scalable, modular extension and composition of large software frameworks. JX/MI builds on previous work on the language JX, which supported scalable extension of software frameworks through *nested inheritance* [31]. Other language mechanisms such as virtual classes [27], open classes [9], and aspects [24] also give some ability to extend a body of code with varying degrees of expressiveness and modularity. What JX/MI adds is the ability to *compose* several extensions to obtain a software system that combines their functionality.

The Polyglot compiler framework is an example of a software framework designed to be extended in sophisticated ways [33]. Polyglot is a Java compiler front end that can be extended to support domain-specific extensions to the Java language. For a given application domain, one might like to choose useful language features from a "menu" of available options, then compose the corresponding compilers to obtain a compiler for the desired language. However, previous language mechanisms gave no way to accomplish this composition while resolving the conflicts among the different extensions.

We identify the following requirements for extension and composition of software systems:

1. Orthogonal extension of the system with both new data types and new operations.
2. Modularity: The base system can be extended without modifying or recompiling its code.
3. Type safety: extensions cannot create run-time type errors.
4. Scalability: extensions should be *scalable*. The amount of code needed should be proportional to functionality added.
5. Non-destructive extension: The base system should still be available for use within the extended system.
6. Composability of extensions.

The first three of these requirements correspond to Wadler's *expression problem* [45]. The fourth requirement, scalability, is often but not necessarily satisfied by supporting separate compilation; it is important for extending large software. Non-destructive extension enables existing clients of the base system and also the extended system itself to interoperate with code and data of the base system, an important requirement for ensuring backward compatibility. Simple nested inheritance [31] addresses the first five requirements, but it does not support extension composition. In JX/MI, composition is accomplished through a new language feature, *multiple nested inheritance*, which builds on nested inheritance by adding the ability to write software by inheriting from several frameworks and thereby obtaining a composition of their functionality.

This paper describes multiple nested inheritance in the JX/MI language and our experience using it to compose software. Section 2 defines the problem of scalable extension and composition, considers a particularly difficult instantiation of this problem—the extension and composition of compilers—and gives an informal introduction to multiple nested inheritance and JX/MI. Multiple nested inheritance creates several interesting technical challenges, such as the problem of resolving conflicts among inherited frameworks; this topic and a detailed informal discussion of language semantics are presented in Section 3, explaining how conflicts are resolved among inherited frameworks. Section 4 then shows an example of using multiple nested inheritance in the construction of an extensible, composable compiler. The implementation of JX/MI is described in Section 5, and Section 6 describes experience using JX/MI to implement and compose extensions in the Polyglot compiler framework and in the Pastry framework for building peer-to-peer systems. Related work is discussed in Section 7, and the paper concludes in Section 8.

2. Multiple nested inheritance for composition

Multiple nested inheritance supports scalable extension of a base system and composition of those extensions. To illustrate how multiple nested inheritance achieves this goal, we will consider the example of building a compiler with composable extensions. A compiler is of course not the only system for which extensibility is useful; other examples include user interface toolkits, operating systems, game engines, web browsers, and peer-to-peer networks. However, compilers are a particularly challenging domain because a compiler has several different interacting dimensions along which it can be extended.

2.1 Nested inheritance

Nested inheritance [31] is a mechanism that allows modular, scalable extension of a large body of code with new functionality. Nested inheritance was introduced in JX, an extension of the Java programming language; JX/MI extends JX with multiple nested inheritance. We begin by reviewing how nested inheritance supports extensibility.

Nested inheritance in JX/MI is inheritance of *namespaces*: that is, classes or packages. A package may contain several classes and packages, and a class may contain nested classes as well as methods and fields. A namespace may extend another namespace, inheriting all its members, including nested namespaces. As with ordinary inheritance, the meaning of code inherited from the base namespace is as if it were copied down from the base. A derived namespace may *override* any of the members it inherits, including nested classes and packages. Like virtual classes [26, 27, 16], when a nested namespace $T.C$ is overridden in T', a derived namespace of T, the overriding namespace $T'.C$ does not replace $T.C$, but instead refines (or *further binds*) it: $T'.C$ inherits members from $T.C$ as well as from $T'.C$'s explicitly named base namespaces (if any). In addition, $T'.C$ is also a subtype of $T.C$.

To illustrate features of JX/MI, Figure 1 shows a fragment of a simple compiler for the lambda calculus extended with pair expressions. This compiler translates the lambda calculus with pairs into the lambda calculus without pairs. In this example, the `pair` package extends the `base` package, further binding the `Visitor` and `Compiler` classes. The class `pair.Visitor` is a subclass of `base.Visitor` and extends it with a `visitPair` method.

Type reinterpretation. The key feature of nested inheritance that enables scalable extensibility is late binding of type names. Type names are resolved in their *inheriting context*: When the name of a class or package is inherited into a new namespace, the name is interpreted in its new context, rather than in the context in which it occurs in the program text. Thus, the meaning of class and package names depends on the run-time value of `this`.

Here, the class `base.Emitter` extends `Visitor`. In the context of the `base` package—that is, in code where `this` is an instance of a class in `base`—the name "`Visitor`" is interpreted as `base.Visitor`. In the context of `pair`, which inherits the declaration of `Emitter` from `base`, "`Visitor`" refers to `pair.Visitor`. Thus, `pair.Emitter` extends `pair.Visitor` and inherits the `visitPair` method of `pair.Visitor`. Reinterpretation of supertype declarations provides a form of *virtual superclasses* [27, 12], permitting the subtyping relationships among the nested namespaces to be preserved when inherited into a new enclosing namespace. The class hierarchy in the original namespace is implicitly replicated in the derived namespace; nested inheritance thus provides a form of *higher-order hierarchy* [15]. Using virtual superclasses, when a class is further bound, new members added into it are automatically inherited by subclasses in the new hierarchy; virtual superclasses thus provide functionality similar to *mixins* [3, 17]. JX/MI also allows the `extends` declaration of a class or interface to be overridden to extend a subtype of the original supertype.

This type name reinterpretation occurs with the formal parameter of the method `accept` as well. The class `pair.Pair` overrides the method `accept` of `base.Exp` with parameter type `Visitor`, which in this context means `pair.Visitor`. The method `pair.Pair.accept` can therefore access the parameter's `visitPair` method. Even though `pair.Visitor` is a subtype of `base.Visitor`, reinterpreting `Visitor` in contravariant positions such as formal parameter types is statically type-safe [31].

Mutually dependent classes. Compilers are composed of sets of related, mutually dependent classes. For example, the classes `Exp` and `Visitor` in the `base` namespace refer to one another. Extending one class at a time, as in ordinary class inheritance, does not work because the extended classes need to know about each other. With ordinary inheritance, the `pair` compiler could define `Pair` as a new subclass of `Exp`, but references within `Exp` to class `Visitor` would refer to the old version of `Visitor`, not the appropriate one that understands how to visit pairs.

By grouping related classes into a namespace, the entire set of classes may be extended at once. Late binding of type names in a namespace ensures that relationships between classes in the original namespace are preserved when these classes are inherited into the new namespace.

In general, some references to other types should be reinterpreted, while others should continue to refer to the same class. Nested inheritance introduced two forms of types to give classes an expressive way to name each other so that references from one class to another are reinterpreted correctly.

A *dependent class* p`.class` is a type representing the run-time class of a given object named p. Classes indexed by a given dependent class belong to the same *family*. In general, the particular set of classes comprising a family may not be known statically and is determined at run time from an object instance. The type system guarantees that classes of the same name but in different families are not accidentally confused. This capability, called *family polymorphism* [14], ensures the static type safety of the extended system by preventing it from treating classes belonging to the base system or to other extensions as if they belonged to the extension. Virtual classes, mixin layers [41], and nested inheritance all provide some form of family polymorphism.

Nested inheritance introduced *prefix types* to permit a class to refer to other classes in the same family that are not nested within it. Prefix types allow any instance of a class in a family to identify the family. In contrast, with virtual classes a single object—for instance, the compiler object—identifies the family, and only classes nested within that object are members of the family. This representative object must be accessible throughout the system.

The expressiveness of prefix types makes it possible in JX and JX/MI to extend a single class that is part of a family of related classes, without extending all of them. Inheritance can operate at every level of the containment hierarchy, enabling fine-grained reuse of individual classes in other namespaces. Related extensibility mechanisms such as virtual classes and mixin layers do not support this.

```
package base;

abstract class Exp {
  Type type;
  abstract Exp accept(Visitor v);
}
class Abs extends Exp {
  String x; Exp e; // λx.e
  Exp accept(Visitor v) {
    e = e.accept(v);
    return v.visitAbs(this);
  }
}
class Visitor {
  Exp visitAbs(Abs a) {
    return a;
  }
}
class TypeChecker extends Visitor
{
  Exp visitAbs(Abs a) { ... }
}
class Emitter extends Visitor {
  Exp visitAbs(Abs a) {
    print(...);
  }
}
class Compiler {
  void main() { ... }
}
```

```
package pair extends base;

class Pair extends Exp {
  Exp fst, snd;
  void accept(Visitor v) {
    fst.accept(v); snd.accept(v);
    return v.visitPair(this);
  }
}
class Visitor {
  Exp visitPair(Pair p) { return p; }
}
class TranslatePairs extends Visitor
{
  Exp visitPair(Pair p) {
    return ...;
    // (λx.λy.λf. f x y) ⟦p.fst⟧ ⟦p.snd⟧
  }
}
class Compiler {
  void main() {
    Exp e = parse();
    e.accept(new TypeChecker());
    e = e.accept(new TranslatePairs());
    e.accept(new Emitter());
  }
}
```

Figure 1. Lambda calculus + pairs compiler

2.2 Extensibility requirements

Nested inheritance meets the first five of the requirements described in Section 1, making it a useful language for implementing extensible systems such as compiler frameworks:

Type-safe orthogonal extension. Compiler frameworks must support the addition of both new data types (abstract syntax, types, dataflow analysis values) and operations on those types (the compiler passes that transform and rewrite these data types). It is well known that there is a tension between extending types and extending the procedures that manipulate them [37]. Nested inheritance solves this problem because reinterpretation of type names in the inheriting context causes inherited methods to operate automatically on the data types as defined in the inheriting context.

Modularity and scalability. Extensions are subclasses (or subpackages) and hence are modular. Nested inheritance is also type-safe [31]. Extension is scalable for several reasons; one important reason is that the name of every method, field, and class provides a potential hook that can be used to extend behavior and data representations.

Nested inheritance does not affect the inherited code, so it is a non-destructive extension mechanism, unlike open classes [9] and aspects [24]. Therefore, inherited code and extended code can be used together in the same system, which is important in extensible compilers because the base language is often used as a target language in an extended compiler.

2.3 Composition

To support composition of extensions, JX/MI extends JX with multiple nested inheritance. Both classes and packages may be composed using multiple inheritance. Suppose that we had also extended the `base` package of Figure 1 to a `sum` package implementing a compiler for the base language extended with sum types. In that case, multiple nested inheritance permits the `pair` and `sum` packages to be composed as shown in Figure 2.

The package `pair_and_sum` inherits all members of `pair` and `sum`. When two namespaces are composed, their common nested namespaces are also composed and their own nested namespaces are preserved. Since both `pair` and `sum` contain a class `Visitor`, the new class `pair_and_sum.Visitor` extends both `pair.Visitor` and `sum.Visitor`. By integrating the member classes of all its base classes, `pair_and_sum` compiler supports both product and sum types.

Both `pair.Compiler` and `sum.Compiler` define a method `main`; `pair_and_sum` resolves the conflict by overriding the `main` method, which is necessary in any case to define the order of compiler passes.

JX/MI provides only *shared* multiple inheritance: when a subclass (or subpackage) extends multiple base classes, perhaps implicitly by inheriting from multiple containing namespaces, the new subclass may share a common superclass of its immediate superclasses; however, instances of the subclass will not contain multiple subobjects for the common superclass. In our example, `pair_and_sum.Visitor` inherits from `base.Visitor` only once, like C++ virtual base classes.

We describe the semantics of multiple inheritance in more detail in the next section.

3. JX/MI

This section gives an overview of the static and dynamic semantics of JX/MI. To conserve space, not all the features of the JX/MI language are discussed in detail; package inheritance, in particular, is discussed only briefly.

3.1 Dependent classes and prefix types

In Figure 1, the name `Visitor` is syntactic sugar for the type `base[this.class].Visitor`. The type `this.class` is a dependent class, representing the statically unknown, but fixed, run-time class of the variable `this`. In general, the dependent class p`.class`

```
package sum extends base;

class Case extends Exp {
  Exp test, ifLeft, ifRight; ...
}
class Visitor {
  Exp visitCase(Case c) {
    return c;
  }
}
class TypeChecker extends Visitor
  { ... }
class TranslateSums extends Visitor
  { ... }
class Compiler {
  void main() { ... }
}
```

```
package pair_and_sum extends pair, sum;

// Resolve conflicting versions of main.
class Compiler {
  void main() {
    Exp e = parse();
    e.accept(new TypeChecker());
    e = e.accept(new TranslatePairs());
    e = e.accept(new TranslateSums());
    e.accept(new Emitter());
  }
}
```

Figure 2. Compiler composition

represents the run-time class of the object referred to by the *final access path p*. A final access path is either a final local variable, including `this` and final formal parameters; a field access *p*.`f`, where *p* is a final access path and `f` is a final field of *p*; or a newly-constructed object `new` *T*`()`. The run-time class of the object specified by a final access path does not change.

The *prefix package* `base[this.class]` represents the enclosing package of `this.class` that is a subpackage of `base`. In the context of `pair` and its members, `base[this.class]` resolves to `pair`. More generally, the *prefix type* *P*[*T*] represents the innermost enclosing namespace of *T* that is a subtype of the non-dependent namespace *P*. Prefix types provide an unambiguous way to name enclosing classes and packages without the overhead of storing references to enclosing instances in each object, as is done in virtual classes. Indeed, if the enclosing namespace is a package, there are no run-time instances that could be used for this purpose.

Dependent classes and prefix types behave like any other class in JX/MI. In particular, instances of these types may be allocated, their static methods and fields may be accessed, and classes and packages may be selected from them.

Both dependent classes and prefixes of dependent classes are *exact types* [4]. Their run-time class is fixed, but statically unknown in general. Simple types like `Visitor` are not exact since variables of this type may contain instances of any subtype of `Visitor`. To make type-checking decidable, a class may not extend an exact type, although it may extend a nested class of an exact type, as is done, for instance, in Figure 1 by the `Abs` class, which extends `base[this.class].Exp`. In addition, dependencies between the types of formal parameters of a method must be acyclic, and a dependent class cannot be dependent upon a field path whose declared type is also a field-dependent class. These restrictions prevent dependency cycles among dependent classes.

As a special case, the dependent class `this.class` may be used in a static context, even though `this` is not in scope. In a static context such as a superclass declaration, `this.class` represents the namespace into which the code containing `this.class` is inherited.

To improve expressiveness and ease porting of Java programs to JX/MI, a *non-final* local variable *x* used in a method call may be coerced to *x*`.class` if *x* is not modified by any of the call's actual arguments, including the receiver. This condition ensures that on entry to the method, all types dependent on *x* are consistent; that is, the run-time class of *x* does not change between the time *x* is evaluated and method entry. Similar rules are used to coerce non-final variables in constructor calls, in `new` expressions, and in field assignments. The optimization is not performed for field paths since it is not safe for multi-threaded programs.

3.2 Type substitution

A method may have a formal parameter whose type depends upon another, including `this`. Such a method may only be called with a corresponding actual argument whose type depends upon another argument. For example, the class `base.Abs` in Figure 1 contains the following call:

`v.visitAbs(`$\texttt{this}_\texttt{A}$`);`

to a method of `base.Visitor` with the signature:

`void visitAbs(base[`$\texttt{this}_\texttt{V}$`.class].Abs a);`

For clarity, each occurrence of `this` has been labeled with an abbreviation of its declared type.

To permit expressions that are not final access paths to be used as actual arguments, the type checker substitutes the actual argument types for dependent classes occurring in the formal parameter types. In this example, the receiver `v` has the type `base[`$\texttt{this}_\texttt{A}$`.class].Visitor`. Substituting this type for $\texttt{this}_\texttt{V}$`.class` in the formal parameter type `base[`$\texttt{this}_\texttt{V}$`.class].Abs` yields `base[base[`$\texttt{this}_\texttt{A}$`.class].Visitor].Abs`, which is equivalent to `base[`$\texttt{this}_\texttt{A}$`.class].Abs`.

To ensure soundness, substitution must satisfy the requirement that *exactness be preserved*; that is, when substituting into an exact type—a dependent class or a prefix of a dependent class—the resulting type must also be exact. This ensures that the run-time class or package represented by the type remains fixed. The substitution above is permitted since both `base[`$\texttt{this}_\texttt{V}$`.class]` and `base[`$\texttt{this}_\texttt{A}$`.class]` are exact.

However, if the type of `v` were just `base.Visitor`, then `v` might refer at run time to a `pair.Visitor` while at the same time $\texttt{this}_\texttt{A}$ refers to a `base.Abs`. Substitution would yield `base[base.Visitor].Abs`, which simplifies to `base.Abs`. Since `base[`$\texttt{this}_\texttt{A}$`.class].Abs` is a subtype of `base.Abs`, the call would, incorrectly, be permitted, leading to a potential run-time type error. The problem is that there is no guarantee that the run-time classes of $\texttt{this}_\texttt{A}$ and `v` both have the same enclosing `base` package. By requiring exactness be preserved, the substitution is illegal since `base` is not exact; therefore, the call to `visitAbs` where `v` is declared to be a `base.Visitor` is not permitted.

```
class A {                   class A2 extends A {
  class B { }                 class B { }
  void m() { }                class C { }
}                             void m() { }
                              void p() { }
class A1 extends A {        }
  class B { }
  class C { }               abstract class D
  void m() { }                extends A1, A2 { }
  void p() { }
}
```

Figure 3. Multiple inheritance with name conflicts

3.3 Intersection types

Multiple inheritance in JX/MI is implemented using intersection types [38, 10]. Rather than extend multiple superclasses, say T_1 and T_2, a class extends a single, implicit *intersection class type* $T_1 \& T_2$, which is a subclass (and subtype) of both T_1 and T_2. The declaration "`class D extends A1, A2`" is sugar for "`class D extends A1 & A2`".

The intersection type inherits all members of its base classes; however, its members cannot be overridden by the intersection class itself since the intersection is not declared explicitly in the program text; subclasses of the intersection type may override members.

When two namespaces declare members with the same name, a *name conflict* may occur. How the conflict is resolved depends on where the name was introduced and whether the name refers to a nested class or to a method. If the name was introduced in a common ancestor of the intersected namespaces, the members with that name are assumed to be related. Otherwise, the name is assumed to refer to distinct members that coincidentally have the same name, but with possibly different semantics.

When two namespaces are intersected, their commonly named nested namespaces are also intersected. In the code in Figure 3, both `A1` and `A2` contain a nested class `B` inherited from `A`. Since a common ancestor of `A1` and `A2` introduces `B`, the intersection type `A1 & A2` contains a nested class (`A1 & A2`).`B`, which is equivalent to `A1.B & A2.B`. The subclass `D` has an implicit nested class `D.B`, a subclass of (`A1 & A2`).`B`.

On the other hand, `A1` and `A2` both declare independent nested classes `C`. Even though these classes have the same name, they may have different semantics. The class (`A1 & A2`).`C` is *ambiguous*. In fact, `A1 & A2` contains two nested classes named `C`, one that is a subclass of `A1.C` and one a subclass of `A2.C`. Class `D` and its subclasses can specify which `C` is meant by exploiting the prefix type notation to resolve the ambiguity: `A1[D].C` refers to the `C` from `A1`, and `A2[D].C` refers to the `C` from `A2`. References to `C` within `A1` are interpreted as `A1[this.class].C`, and when inherited into `D`, these references refer to the `C` inherited from `A1`. Similarly, references to `C` within `A2` inherited into `D` refer to the `C` inherited from `A2`.

A similar situation occurs with the methods `A1.p` and `A2.p`. Again, `D` inherits both versions of `p`. Callers of `D.p` must resolve the ambiguity by up-casting the receiver to specify which one of the methods to invoke.

Finally, two or more intersected classes may declare methods with the same signature that override a method declared in a common base class. In this case, illustrated by the method `m` in Figure 3, the method in the intersection type is considered *abstract*. Subclasses of the intersection type (`D`, in the example), must override `m` to resolve the conflict, or else also be declared abstract.

Dependent classes and prefix types impose some restrictions on which types may be intersected. Intersecting two different exact types is not permitted, because members of the intersection type would be objects with more than one run-time class, which is impossible. Since intersection types are used only in `extends` declarations, where only `this` for the immediately enclosing class is in scope, this restriction does not limit expressiveness in practice. Similarly, a dependent type cannot be intersected with a non-dependent type unless the non-dependent type is a supertype of the dependent type (in which case, the intersection type is equivalent to just the subtype). Finally, two types conflict if any prefix of the types conflict; hence, subclasses of their intersection must be declared abstract. This restriction ensures that uses of prefix types inherited into the intersection type are meaningful.

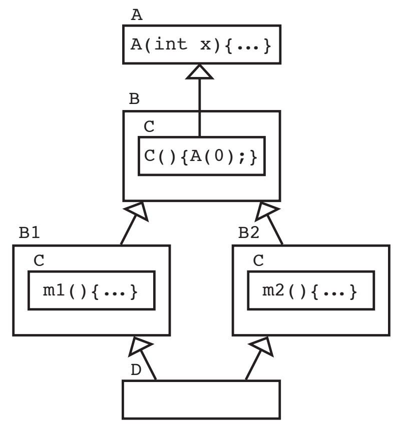

Figure 4. Constructors in implicitly shared classes

3.4 Constructors

As in Java, JX/MI initializes objects using constructors. JX/MI permits instances of dependent types to be allocated. Since the class being allocated may not be statically known, JX/MI allows the programmer to invoke a constructor of a superclass of the type being allocated. Constructors in JX/MI are inherited and can be overridden similarly to methods, ensuring the class represented by a dependent type implements a constructor with the same signature as the constructor invoked. The programmer can prevent a constructor from being inherited by declaring it `nonvirtual`; it is illegal to invoke a `nonvirtual` constructor on a dependent class.

A constructor for a given class must specify, using super constructor calls, how to invoke a constructor of each of the class's declared immediate superclasses. When a class explicitly extends multiple superclasses, it may share a common superclass. Invoking the shared constructor more than once may lead to inconsistent initialization of `final` fields, admitting the possibility of a run-time type error if the fields are used in dependent types.

To prevent this situation, if explicit multiple inheritance introduces sharing, JX/MI requires the new subclass (which introduced the sharing) to explicitly invoke a constructor of the shared superclass. The super constructor calls in the immediate subclasses of the shared class are not evaluated. This behavior is similar to that of constructors of shared virtual base classes in C++.

Sharing can also be introduced implicitly. For example, in Figure 4, the implicit class `D.C` is a subclass of `B1.C & B2.C` and shares the superclass `A`. Since `B1.C` and `B2.C` both inherit their `C()` constructor from `B.C`, both inherited constructors invoke the `A` constructor with the same arguments. There is no conflict and the compiler need only ensure that the constructor of `A` is invoked exactly once, before the body of `D.C`'s constructor is executed.

If, on the other hand, one or both of `B1` and `B2` overrode the `C()` constructor, then since `B1.C` and `B2.C` have different constructors

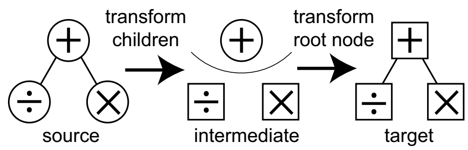

Figure 5. AST transformation

with the same signature, one of them might change how the `C` constructor invokes `A(int)`. There is a conflict and `D` must further bind `C` to specify how `C()` should invoke the constructor of `A`.

3.5 Extensible rewriters

One challenge for building extensible software systems is to provide extensible data processing, particularly when the input and output data have complex structure. Extensions to the software need to be able to scalably and modularly extend both the transformations performed on the data and the data being transformed. Compilers exhibit this difficulty, because compiler passes perform complex transformations on complex data structures representing program code. For scalable extensibility, it should not be necessary to change data transformers (e.g., compiler passes) if the extensions to the data representation do not interact with the transformation in question.

A partial solution to this problem is the Visitor design pattern [18], which supports scalable extension of data processing. It allows boilerplate traversal of the input data structure to be factored out and shared. With minor extensions, visitors also support the generation of structured output data.

For example, consider an abstract syntax tree (AST) node representing a binary operation. As illustrated in Figure 5, most compiler passes for this kind of node would recursively transform the two child nodes representing operands, then invoke pass-specific code to transform the binary operation node itself, in general constructing a new node using the new children. This generic code can be shared by many passes.

However, the code for a given compiler pass might not be aware of the particular extended AST form used by a given compiler extension, and in general the source and target of the pass may be AST nodes for different languages—both, perhaps, extended versions of the base AST representation that the pass operates on. Because it is unaware of any new children of the node added by extensions of the source language of the pass, it is hard to write a reusable compiler pass; the pass may fail to transform all the node's children. In the compiler example of Figure 1, a compiler pass transforms expressions in the lambda calculus extended with pairs into lambda calculus expressions without pairs. If this compiler pass is reused in a compiler in which expressions have additional type annotations, the source and target languages node will have children for these additional annotations, but the pass will not be aware of them and will fail to transform them.

To make a pass aware of any new children added by extensions of the source language, while preserving modularity, the solution is for the compiler to represent nodes in the intermediate form as trees with a root in the source language and children in the target language, corresponding to the middle tree of Figure 5. In the example of Figure 1, this can be done by creating, for both the source (i.e., `pair`) and target (i.e., `base`) language, packages `ast_struct` defining just the structure of each AST node. The `ast_struct` package is then extended to create source and target language packages for the actual AST nodes, and also to create a package *inside each visitor class* for the intermediate form nodes of that visitor's specific source and target language. This design is shown in Figure 6.

The key to making this design type-safe is a variant of virtual types [26, 27]—in this case, *virtual packages*, to link the packages together. In the `ast_struct` package, children of each AST node reside in a `child` virtual package. Like virtual types, virtual packages can be further bound in subclasses and subpackages. The `ast` package extends the `ast_struct` package and overrides `child` to bind it to the same `ast` package itself; the node classes in `ast` have children in the same package as their parent.

The `Visitor.tmp` package also extends the `ast_struct` package, but overrides `child` to bind it to the `target` package, which represents the target language of the visitor transformation. AST node classes in the `tmp` package have children in the `target` package, but parent nodes are in the `tmp` package; since `tmp` is a subpackage of `ast_struct`, nodes in this package have the same structure as nodes in the visitor's sibling `ast_struct` package. Thus, if the `ast_struct` package is overridden to add new children to an AST node class, the intermediate nodes in the `tmp` package will also contain those children.

Virtual types in JX/MI differ from those in languages from the BETA [26] family. First, virtual types in JX/MI are attributes of classes rather than of objects. Second, to enforce the requirement that exactness be preserved by substitution (see Section 3.2), virtual types and packages can be declared *exact*. For a given run-time container namespace T, the exact virtual type $T.C$ must be a fixed run-time class. Unlike a final-bound virtual type [26], an exact virtual type can be overridden in a subclass. For example, consider these declarations:

```
class A { }
class A2 extends A { }
class B { exact class T = A; }
class B2 extends B { exact class T = A2; }
```

The exact virtual type `B.T` is equivalent to the dependent class `new A().class`; that is, `B.T` contains only instances with run-time class `A`. Similarly, `B2.T` is equivalent to `new A2().class`. If a variable `b` has declared type `B`, then an instance of `b.class.T` may have run-time class either `A` or `A2`, depending on the run-time class of `b`.

3.6 Packages

JX/MI supports inheritance of packages, including multiple inheritance. In fact, the most convenient way to use nested inheritance is usually at the package level, because large software is usually contained inside packages, not inside classes. Packages are treated like classes that contain no fields, methods, or constructors. The semantics of prefix packages and intersection packages are similar to those of prefix and intersection class types, described above. Since packages do not have run-time instances, the only exact packages are prefixes of a dependent class nested within the package.

4. Composing compilers

Using the language features just described we can construct a composable, extensible compiler.

Scalable, orthogonal extension of the base compiler with new data types and new operations is achieved through nested inheritance. Type name reinterpretation, made type-safe through dependent classes and prefix types, allows the entire base compiler to be extended and individual classes overridden to provide new functionality.

Because supertype declarations are interpreted in their inheriting context, new methods, fields, and member classes, added into an overridden class are automatically inherited into its subclasses. Overriding classes introduces entirely new versions of those classes. The original base classes are still available and need not be recompiled.

```
package base.ast_struct;

exact package child = ast_struct;
abstract class Exp { }
class Abs extends Exp {
  String x; child.Exp e;
}
```

```
package base.ast extends ast_struct;

exact package child
  = base.ast[this.class];
abstract class Exp {
  abstract v.class.target.Exp
    accept(Visitor v);
  void childrenExp(Visitor v,
    v.class.tmp.Exp t) {
  }
}
```

```
package base;

class Visitor {
  // source language
  //   = base[this.class].ast
  // target language
  //   <= base.ast;
  exact package target = base.ast;
  package tmp extends ast_struct {
    exact package child = target;
  }
  ...
}
```

Figure 6. Extensible rewriting example

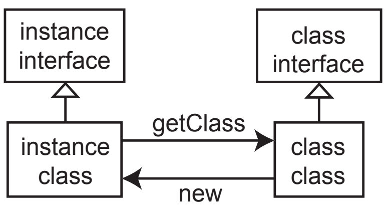

Figure 7. Target classes and interfaces

To support new syntax, an extended compiler need only introduce new abstract syntax node classes. If the base compiler uses the visitor design pattern, the base visitor interface can be extended with `visit` methods for the new nodes implementing default behavior, such as an identity transformation. Visitors for passes affected by the new syntax can be overridden to support it.

New passes can be added to the compiler by creating new visitor classes or by adding methods to the abstract syntax classes. Here again, default behavior can be added into the root class of all AST classes, and only those node types affected by the new pass need be overridden.

Independent compiler extensions can be composed using multiple nested inheritance with minimal effort. If the two compiler extensions are orthogonal, as for example with the product and sum type compilers of Section 2.3, then composing the extensions is trivial. If the language extensions have conflicting semantics, this will manifest as a name conflict when intersecting the classes within the two compilers. These name conflicts must be resolved to be able to instantiate the composed compiler.

5. Implementation

We implemented the JX/MI compiler in the Polyglot framework [33]. The compiler is a 2700-LOC (lines of code) extension of the JX compiler [31], itself a 22-kLOC extension of the Polyglot base Java compiler. (Note: blank and comment lines are not counted.)

5.1 Translating classes

The translation of JX/MI to Java is similar to the translation of JX to Java source code, described in [31]. As illustrated in Figure 7, each JX/MI class is represented by four classes: an *instance class*, an *instance interface*, a *class class*, and a *class interface*. The translation scheme described in the earlier paper has been extended to support intersection types and to make the generated code cleaner and more efficient.

References to a class or interface `C` are translated to references to `C`'s *instance interface*. The instance interface of `C` contains signatures for all instance methods of `C` as well as field getters and setters to allow access to fields from contexts where the actual run-time class is unknown. The instance interface of `C` extends the instance interface of all of `C`'s supertypes. Dependent classes, prefix types, and virtual types are translated to the instance interface of their most precise statically known simple supertype.

At run time, an object of the JX/MI class `C` is represented as a single object of the *instance class*, which implements the method interface of `C`. Instance classes are also generated for intersection classes. For non-intersection classes, the instance class of `C` extends the instance class of `C`'s explicit superclass, which may be an intersection class. The instance class of an intersection class extends the instance class of the common ancestor of the intersected classes. Such an ancestor exists, since all classes are subclasses of `Object`. An instance class for `C` contains, or inherits from its superclass, all fields declared in `C` or inherited from any of the superclasses of `C`.

For every JX/MI class, there is a singleton *class class* object instantiated at run time. Every instance class contains a reference to its class class. Static methods are translated to instance methods of the class class to allow static methods to be invoked on dependent types, where the actual run-time class is unknown. To support `super` calls in the presence of multiple inheritance, instance methods are translated to methods of the class class. The instance class contains short one-line methods to dispatch to the implementation of the method in the appropriate class class. The class class also provides functions for accessing run-time type information to implement `instanceof` and casts, for constructing instances of the class, and for accessing the class class of prefixes and members classes, including virtual types. The code generated for expressions that dispatch on a dependent class—`new x.class()` expressions, for example—evaluates the dependent class's access path to locate the class class for the type. For prefix types, the class class is used to navigate to the prefix of the type.

The class class implements the *class interface* of each of the class's supertypes. The class interface contains signatures for all static methods of the class and also a factory method for each constructor.

5.2 Translating packages

To support package inheritance and composition, the representation of a package `p` includes a *package interface* and a *package class* that implements the interface, analogous to the class interface and class class. The package class provides type information about the package at run time and access to the class class or package class singletons of its members and prefixes. Both the package class and package interface of `p` are members of package `p`; packages have no instance classes or instance interfaces.

5.3 Java compatibility

Since JX/MI is translated to Java, the generated code can only use single inheritance. To interact with Java code, a JX/MI class

	j0	pao	carray	covarRet
coffer	63	86	34	66
j0		46	34	37
pao			34	53
carray				31

Table 2. Polyglot composition results: lines of code

may have only one most-specific Java superclass. The generated instance class is a subclass of this Java class. Because the instance interface is not a subtype of any Java class (except `Object`), when passing JX/MI objects to a method expecting a Java class, the object must be cast from the instance interface type to the expected Java supertype.

6. Experience

6.1 Polyglot

Following the approach described in Section 4, we ported the Polyglot compiler framework and several Polyglot-based extensions, all written in Java, to JX/MI. The Polyglot base compiler is a 31.9 kLOC program that performs semantic checking on Java source code and outputs equivalent Java source code. Special design patterns make Polyglot highly extensible [31]; more than a dozen research projects have used Polyglot to implement various extensions to Java (e.g., JPred [30], JMatch [25], as well as JX and JX/MI). For this work we ported six extensions ranging in size from 200 to 3000 LOC.

The port of the base compiler was our first attempt to port a large program to JX/MI, and was completed by one of the authors within a few days, excluding time to fix bugs in the JX and JX/MI compilers. Porting of each of the extensions took from one hour to a few days. Much of the porting effort could be automated, with most files requiring only modification of `import` statements. Porting issues are described below.

The ported base compiler is 28.0 kLOC. The code becomes shorter because it eliminates factory methods and other extension patterns necessary to make the Java version extensible, but which are not needed in JX/MI. We eliminated only extension patterns that were obviously unnecessary, and could remove additional code with more effort.

The number of type downcasts in each extension is reduced in JX/MI. For example, `coffer` went from 192 to 102 downcasts. The reduction is due to (1) use of dependent types, obviating the need for casts to access methods and fields introduced in extensions, and (2) removal of old extension mechanism code. Receivers of calls to conflicting methods sometimes needed to be upcast to resolve the ambiguities; there are 19 such upcasts in the port of `coffer`.

The extensions are summarized in Table 1. The parsers for the base compiler, extensions, and compositions were generated from CUP [21] or Polyglot parser generator (PPG) [33] grammar files. Because PPG supports only single grammar inheritance, grammars were composed manually; line counts do not include parser code.

Table 2 shows lines of code needed to compose each pair of extensions, producing working compilers that implemented a composed language. The `param` extension was not composed because it is an *abstract extension* containing infrastructure for parameterized types, and it does not change the language semantics; however, `coffer` extends the `param` extension.

The data show that all the compositions can be implemented with very little code; further, most added code straightforwardly resolves trivial name conflicts, such as between the methods that return the name and version of the compiler. Only three of ten compositions (`coffer` & `pao`, `coffer` & `covarRet`, and `pao` & `covarRet`) required resolution of nontrivial conflicts, for example, resolving conflicting code for checking method overrides. The code to resolve these conflicts is no more 10 lines in each case.

Name	LOC original	LOC ported
Pastry	7082	7363
Beehive	3686	3634
PC-Pastry	695	630
CorONA	626	591
`cache`	N/A	140
CorONA–Beehive	N/A	68
CorONA–PC-Pastry	N/A	28

Table 3. Ported Pastry extensions and compositions

6.2 Pastry

We also ported the FreePastry peer-to-peer framework [39] version 1.2 to JX/MI and composed a few Pastry applications. The sizes of the original and ported Pastry extensions are shown in Table 3. Excluding bundled applications, FreePastry is 7100 lines of Java code.

Host nodes in Pastry exchange messages that can be handled in an application-specific manner. In FreePastry, network message dispatching is implemented with `instanceof` statements and casts. We changed this code to use more straightforward method dispatch instead, thus making dispatch extensible and eliminating several downcasts. Messages are dispatched to several protocol-specific handlers. For example, there is a handler for the routing protocol, another for the join protocol, and others for any applications built on top of the framework. The Pastry framework allows applications to choose to use one of three different messaging layer implementations: an RMI layer, a wire layer that uses sockets or datagrams, and an in-memory layer in which nodes of the distributed system are simulated in a single JVM. Family polymorphism enforced by the JX/MI type system statically ensures that messages associated with a given handler are not delivered to another handler and that objects associated with a given transport layer are not used by code for a different layer implementation.

Pastry implements a distributed hash table interface. Beehive [36] and PC-Pastry extend Pastry with caching functionality. PC-Pastry [36] uses a simple passive caching algorithm, where lookups are cached on nodes along the route from the requesting node to a node containing a value for the key. Beehive actively replicates objects throughout the network according to their popularity. We introduced a package ("`cache`") containing functionality in common between Beehive and PC-Pastry; the CorONA RSS feed aggregation service [35] was modified to extend the `cache` package rather than Beehive.

Using multiple nested inheritance, the modified CorONA was composed first with Beehive, and then with PC-Pastry, creating two applications providing the CorONA RSS aggregation service but using different caching algorithms. Each composition of CorONA and a caching extension contains a single `main` method and some configuration constants to initialize the cache manager data structures. The CorONA–Beehive composition also overrides some CorONA message handlers to keep track of each cached object's popularity. We also implemented and composed test drivers for the CorONA extension, but line counts for these are not included since the original Java code did not include them.

The JX/MI code for FreePastry is 7400 LOC, 300 lines longer than the original Java code. The additional code consists primarily of interfaces introduced to implement network message dispatching. The Pastry extensions had similar message dispatch-

Name	Extends Java 1.4 ...	LOC original	LOC ported
`polyglot`	with nothing	31888	27984
`param`	with infrastructure for parameterized types	513	540
`coffer`	with resource management facilities similar to Vault [11]	2965	2642
`j0`	with pedagogical features	679	436
`pao`	to treat primitives as objects	415	347
`carray`	with constant arrays	217	122
`covarRet`	to allow covariant method return types	228	214

Table 1. Ported Polyglot extensions

ing overhead; since code in common between Beehive and PC-Pastry was factored out into the `cache` extension, the size of the ported extensions is smaller. The size reduction in CorONA is partially attributed to moving code from the CorONA extension to the CorONA–Beehive composition.

6.3 Porting Java to JX/MI

Porting Java code to JX/MI was usually straightforward, but certain common issues are worth discussing.

Type names. In JX/MI, unqualified type names are syntactic sugar for members of `this.class` or a prefix of `this.class`, e.g., `Visitor` might be sugar for `base[this.class].Visitor`. In Java, unqualified type names are sugar for fully qualified names; thus, `Visitor` would resolve to `base.Visitor`. To take full advantage of the extensibility provided by JX/MI, fully qualified type names sometimes must be changed to be only partially qualified.

In particular, `import` statements in most compilation units are rewritten to allow names of other classes to resolve to dependent types. For example, in Polyglot the import statement `import polyglot.ast.*;` was changed to `import ast.*;` so that imported classes resolve to classes in `polyglot[this.class].ast` rather than in `polyglot.ast`.

Final access paths. To make some expressions pass the type checker, it was necessary to declare some variables final to allow them to be coerced to dependent classes. In many cases, non-final access paths used in method calls could be coerced automatically by the compiler, as described in Section 3.1. However, non-final field accesses were not coerced automatically because the field might be updated (possibly by another thread) between evaluation and method entry. The common workaround is to save non-final fields in a final local variable and then to use that variable in the call.

This issue was not as problematic as originally expected. In fact, in 30 kLOC of ported Polyglot code, only three such calls needed to be modified. In most other cases, the actual method receiver type was of the form P`[`p`.class]`.Q and the formal parameter types were of the form P`[this.class]`.R. Even if an actual argument were updated between its evaluation and method entry its new value is a class enclosed by the same run-time namespace P`[`p`.class]` as the receiver, ensuring that the call is safe.

Path aliasing The port of Pastry and its extensions made more extensive use of field-dependent classes than the Polyglot port. Several casts needed to be inserted in the JX/MI code for Pastry to allow a type dependent upon one access path to be coerced to a type dependent upon another path. Often, the two paths refer to the same object, ensuring the cast will always succeed. Implementing a simple local alias analysis should eliminate the need for many of these casts.

7. Related work

There has been great interest in the past several years in mechanisms for providing greater extensibility in object-oriented languages. object-oriented languages with additional extensibility. Nested inheritance uses ideas from many of these other mechanisms to create a powerful and relatively transparent mechanism for code reuse.

Virtual classes. Nested classes in JX/MI are similar to virtual types and virtual classes [26, 27, 22, 16]. Virtual types were originally developed for the language BETA [26, 27], primarily for generic programming rather than for extensibility.

Although virtual types in BETA were not statically type safe, Ernst's generalized BETA (gbeta) language [12, 13] uses path-dependent types, similar to dependent classes in JX/MI, to ensure static type safety. Type-safe virtual classes using path-dependent types were formalized by Ernst et al. [16]. However, virtual classes may only have one enclosing instance; for this reason, a class may not extend a more deeply nested virtual class. This can limit the ability to extend components of a larger system.

Virtual classes in gbeta support *family polymorphism* [14]: two virtual classes enclosed by distinct objects cannot be statically confused. Because nested classes in JX/MI are attributes of their enclosing class, rather than an enclosing object, nested inheritance supports *class-based family polymorphism*. With family polymorphism, each object defines a family of mutually dependent classes; with class-based family polymorphism, each dependent class defines a family. By using prefix types, any member of the family can be used to name the family; with virtual classes all family members must be named from a single "family object".

Scala [34] is another language that supports scalable extensibility and family polymorphism through a statically safe virtual type mechanism based on path-dependent types. However, Scala's path-dependent type p`.type` is a singleton type containing only the value named by access path p; in JX/MI, p`.class` is not a singleton. For instance, `new x.class(...)` creates a new object of type `x.class` distinct from the object referred to by `x`. This difference gives JX/MI more flexibility, while preserving type soundness. Scala has no analogue to prefix types nor does it provide virtual superclasses, mitigating the scalability of the extension mechanisms provided.

Concord [23] also provides a type-safe variant of virtual classes. In Concord, mutually dependent classes are organized into *groups*, which can be extended via inheritance. References to other classes within a group are made using types dependent on the current group, `MyGrp`, similarly to how prefix types are used in JX/MI. Relative supertype declarations provide functionality similar to virtual superclasses. Groups in Concord cannot be nested, nor can groups be multiply inherited.

Class hierarchy composition. Tarr et al. [44] define a specification language for composing class hierarchies. Rules specify how to merge "concepts" in the different hierarchies. Multiple nested inheritance supports composition with a rule analogous to merging concepts by name.

Snelting and Tip [42] present an algorithm for composing class hierarchies and a semantic interference criterion. If the hierarchies are *interference-free*, the composed system preserves the original behavior. JX/MI reports a conflict if composed class hierarchies have a *static interference*, but makes no effort to detect dynamic interference.

Multiple inheritance and mixins. Cardelli [8] presents a formal semantics of multiple inheritance. Intersection types were introduced by Reynolds in the language Forstythe [38] and were used by Compagnoni and Pierce to model multiple inheritance [10].

The distinction between name conflicts among methods introduced in a common base class and among methods introduced independently with possibly different semantics was made as early as 1982 by Borning and Ingalls [2]. Many languages, such as C++ [43] and Self [20], treat all name conflicts as ambiguities to be resolved by the caller. Some languages [28, 3, 40] allow methods to be renamed or aliased.

A *mixin* [3, 17], also known as an *abstract subclass*, is a class parameterized on its superclass. Mixins are able to provide uniform extensions, such as adding new fields or methods, to a large number classes. Mixins can be simulated using explicit multiple inheritance. JX/MI provides additional mixin-like functionality by allowing the superclass of an existing base class to be changed or fields and methods to be added by overriding the class's superclass through extension of the superclass's container. Additionally, nested inheritance allows the implicit subclasses of the new base class to be instantiated without writing any additional code. Mixins have no analogous mechanism.

Since mixins are composed linearly, a class may not be able to access a member of a given super-mixin because the member is overridden by another mixin. Explicit multiple inheritance imposes no ordering on composition of superclasses. Both gbeta [13] and Scala [34] support mixin composition. Similarly to multiple nested inheritance, composition of mixins in these languages combines nested classes.

Self types and matching. Bruce et al. [6, 4] introduce *matching* as an alternative to subtyping, with a *self type*, or `MyType`, representing the type of the method's receiver. The dependent class `this.class` is similar but represents only the class referred to by `this` and not its subclasses. Type systems with `MyType` decouple subtyping and subclassing; in PolyTOIL and LOOM, a subclass *matches* its base class but is not a subtype. With nested inheritance, subclasses are subtypes. In [7, 5], Bruce and Vanderwaart propose *type groups* as a means to aggregate and extend mutually dependent classes, similarly to Concord's group construct.

Open classes. An *open class* [9] is a class to which new methods can be added without needing to edit the class directly, or recompile code that depends on the class. Nested inheritance provides similar functionality through class overriding in an extended container. Nested inheritance provides additional extensibility that open classes do not, such as the "virtual" behavior of constructors, and the ability to extend an existing class with new fields that are automatically inherited by its subclasses.

Classboxes A *classbox* [1] is a module-based reuse mechanism. Classes defined in one classbox may be imported into another classbox and refined to create a subclass of the imported class. By dispatching based on a dynamically chosen classbox, names of types and methods occurring in imported code is late bound to refined versions of those types and methods. This feature provides similar functionality to the type name reinterpretation provided by `this`-dependent classes and prefix types in JX/MI.

Aspect-oriented programming. Aspect-oriented programming (AOP) [24] is concerned with the management of *aspects*, functionality that cuts across modular boundaries. Nested inheritance provides aspect-like extensibility; an extension of a container may implement functionality that cuts across the class boundaries of the nested classes. Aspects modify existing class hierarchies, whereas nested inheritance creates a new class hierarchy, allowing the new hierarchy to be used alongside the old. Caesar [29] is an aspect-oriented language that also supports family polymorphism, permitting application of aspects to mutually recursive nested types.

8. Conclusions

This paper introduces multiple nested inheritance and shows that it is an effective language mechanism for extending and composing large bodies of software. Extension and composition are scalable, because new code needs to be written only to implement new functionality or to resolve conflicts between composed classes and packages. Novel features like prefix types and exact virtual types offer important expressive power.

Multiple nested inheritance has been implemented in an extension of Java called JX/MI. Using JX/MI, we implemented a compiler framework for Java, and showed that different domain-specific compiler extensions can easily be composed, resulting in a way to construct compilers by choosing from available language implementation components. We demonstrated the utility of multiple nested inheritance outside the compiler domain by porting the FreePastry peer-to-peer system to JX/MI. The effort required to port Java programs to JX/MI is not large. Ported programs were smaller, required fewer type casts, and supported more extensibility and composability.

We have informally described here the static and dynamic semantics of JX/MI. We have not shown that this type system is sound; however, it appears feasible to extend the previous proof for the soundness of JX [32].

Multiple nested inheritance is a powerful and convenient mechanism for building highly extensible software. We expect it to be useful for a wide variety of applications.

Acknowledgments

Steve Chong, Jed Liu, Lantian Zheng provided useful feedback on various drafts of this paper. Thanks also to Venugopalan Ramasubramanian for insightful discussions about Pastry and Beehive.

This research was supported in part by ONR Grant N00014-01-1-0968, NSF Grants 0208642, 0133302, and 0430161, and an Alfred P. Sloan Research Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright annotation thereon. The views and conclusions here are those of the authors and do not necessarily reflect those of ONR, the Navy, or the NSF.

References

[1] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proc. OOPSLA '05*, pages 177–189, October 2005.

[2] Alan Borning and Daniel Ingalls. Multiple inheritance in Smalltalk-80. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 234–237, August 1982.

[3] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proc. OOPSLA '90*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[4] Kim B. Bruce. Safe static type checking with systems of mutually recursive classes and inheritance. Technical report, Williams College, 1997. `http://cs.williams.edu/~kim/ftp/RecJava.ps.gz`.

[5] Kim B. Bruce. Some challenging typing issues in object-oriented languages. *Electronic Notes in Theoretical Computer Science*, 82(8):1–29, October 2003.

[6] Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *European Conference on Object-Oriented Programming (ECOOP)*, number 952 in Lecture Notes in Computer Science, pages 27–51. Springer-Verlag, 1995.

[7] Kim B. Bruce and Joseph C. Vanderwaart. Semantics-driven language design: Statically type-safe virtual types in object-oriented languages. In *Mathematical Foundations of Programming Semantics (MFPS), Fifteenth Conference*, volume 20 of *Electronic Notes in Theoretical Computer Science*, pages 50–75, April 1999.

[8] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Also in *Readings in Object-Oriented Database Systems,* S. Zdonik and D. Maier, eds., Morgan Kaufmann, 1990.

[9] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA 2000 Conference on Object-Oriented Programming, Systems, Languages, and Applications, Minneapolis, Minnesota*, volume 35(10), pages 130–145, 2000.

[10] Adriana B. Compagnoni and Benjamin C. Pierce. Higher order intersection types and multiple inheritance. *Mathematical Structures in Computer Science*, 6(5):469–501, 1996.

[11] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

[12] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[13] Erik Ernst. Propagating class and method combination. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 67–91. Springer-Verlag, June 1999.

[14] Erik Ernst. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, LNCS 2072, pages 303–326, Heidelberg, Germany, 2001. Springer-Verlag.

[15] Erik Ernst. Higher-order hierarchies. In *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 303–329, Heidelberg, Germany, July 2003. Springer-Verlag.

[16] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proc. 33th ACM Symp. on Principles of Programming Languages (POPL)*, Charleston, South Carolina, January 2006. To appear.

[17] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 171–183, San Diego, California, 1998.

[18] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, MA, 1994.

[19] Carl Gunter and John C. Mitchell, editors. *Theoretical aspects of object-oriented programming*. MIT Press, 1994.

[20] Urs Hölzle, Bay-Wei Chang, Craig Chambers, Ole Agesen, and David Ungar. *The* SELF *Manual, Version 1.1*, February 1991. Unpublished manual.

[21] Scott E. Hudson, Frank Flannery, C. Scott Ananian, Dan Wang, and Andrew Appel. CUP LALR parser generator for Java, 1996. Software release. Located at `http://www.cs.princeton.edu/∼appel/-modern/java/CUP/`.

[22] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Foundations for virtual types. In *Proceedings of the Thirteenth European Conference on Object-Oriented Programming (ECOOP'99)*, number 1628 in Lecture Notes in Computer Science, pages 161–185. Springer-Verlag, June 1999.

[23] Paul Jolly, Sophia Drossopoulou, Christopher Anderson, and Klaus Ostermann. Simple dependent types: Concord. In *ECOOP Workshop on Formal Techniques for Java Programs (FTfJP)*, June 2004.

[24] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 220–242, Jyväskylä, Finland, June 1997. Springer-Verlag.

[25] Jed Liu and Andrew C. Myers. JMatch: Abstract iterable pattern matching for Java. In *Proc. 5th Int'l Symp. on Practical Aspects of Declarative Languages (PADL)*, pages 110–127, New Orleans, LA, January 2003.

[26] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*. Addison-Wesley, June 1993.

[27] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Proc. OOPSLA '89*, pages 397–406, October 1989.

[28] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.

[29] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100, March 2003.

[30] Todd Millstein. Practical predicate dispatch. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, October 2004.

[31] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 99–115, October 2004.

[32] Nathaniel Nystrom, Stephen Chong, and Andrew C. Myers. Scalable extensibility via nested inheritance. Technical Report 2004–1940, Computer Science Dept., Cornell University, June 2004.

[33] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *Compiler Construction, 12th International Conference, CC 2003*, number 2622 in Lecture Notes in Computer Science, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag.

[34] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. OOPSLA '05*, pages 41–57, October 2005.

[35] Venugopalan Ramasubramanian, Ryan Peterson, and Emin Gun Sirer. Corona: A high performance publish-subscribe system for the World Wide Web. In *Proceedings of Networked System Design and Implementation (NSDI)*, May 2006.

[36] Venugopalan Ramasubramanian and Emin Gün Sirer. Beehive: $O(1)$ lookup performance for power-law query distributions in peer-to-peer overlays. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, March 2004.

[37] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages*, pages 157–168. Institut de Recherche d'Informatique et d'Automatique, Le Chesnay, France, 1975. Reprinted in [19], pages 13–23.

[38] John C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS-96-146, Carnegie Mellon University, June 1996.

[39] Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[40] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In Luca Cardelli, editor, *Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 248–274, Darmstadt, Germany, July 2003. Springer-Verlag.

[41] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings ECOOP'98*, pages 550–570, Brussels, Belgium, 1998.

[42] Gregor Snelting and Frank Tip. Semantics-based composition of class hierarchies. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 562–584, Málaga, Spain, 2002. Springer-Verlag.

[43] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[44] Peri Tarr, Harold Ossher, William Harrison, and Jr. Stanley M. Sutton. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE)*, pages 107–119, May 1999.

[45] Philip Wadler and et al. The expression problem, December 1998. Discussion on Java-Genericity mailing list.

[46] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

programs	Pr	$::= \langle \overline{L}, e \rangle$
class declarations	L	$::= \texttt{class}\ C\ \texttt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}$
field declarations	F	$::= [\texttt{final}]\ T\ f = e$
method declarations	M	$::= T\ m(\overline{T}\ \overline{x})\ \{e\}$
types	T	$::= C \mid T.C \mid p.\texttt{class} \mid P\texttt{[}T\texttt{]} \mid \texttt{\&}\overline{T}$
non-dependent types	S	$::= C \mid S.C \mid P\texttt{[}S\texttt{]} \mid \texttt{\&}\overline{S}$
classes	P	$::= C \mid P.C$
values	v	$::= \texttt{null} \mid \ell_S$
access paths	p	$::= v \mid x \mid p.f$
expressions	e	$::= v \mid x \mid e.f \mid e_0.f = e_1$
		$\mid e_0.m(\overline{e}) \mid \texttt{new}\ T(\overline{f} = \overline{e}) \mid e_1;\ e_2$
typing environments	Γ	$::= \emptyset \mid \Gamma, x : T$

Figure 8. Grammar

A. Formal semantics

This section presents a formal semantics for the core JX/MI type system and states a soundness theorem for the semantics. To reduce complexity and to save space, several features including package inheritance, constructors, and virtual types are not modeled in the semantics.

A grammar for the calculus is shown in Figure 8. We use the notation $\overline{a}$ for the set $a_1, \ldots, a_n$ for $n \geq 0$. A term with a list subterm should be interpreted as a list of terms; for example, $\overline{T}\ \overline{x}$ should be read $T_1\ x_1, \ldots, T_n\ x_n$.

Programs Pr consist of a set of class declarations $\overline{L}$ and a "`main`" expression e. To avoid cluttering the semantics, we assume a fixed program Pr; all inference rules are implicitly parameterized on Pr. A class declaration L contain a class name C, a superclass declaration T and member classes, fields, and methods $\overline{L}$, $\overline{F}$, $\overline{M}$, respectively. A field declaration F may be final or non-final and consists of a type, field name, and default initializer expression. Methods M have a return type, formal parameters, and method body.

Types T are either names of top-level classes C, nested classes $T.C$, dependent classes p`.class`, prefix types P`[`T`]`, or intersection types `&`$\overline{T}$. Non-dependent types are written S and class names are written P. In the calculus, the prefix type P`[`T`]` is well-formed only if T has a superclass that is immediately enclosed by a subclass of P. More general prefix types can be constructed by desugarring to this form; for example if `c` has type `A.B.C`, then `A[c.class]` desugars to `A[A.B[c.class]]`.

A value is either `null` or a location ℓ_S, which maps to an object on the heap of type S. A final access path p is either a value, a parameter x, or a field access $p.f$. Expressions are values, parameters x, field accesses, field assignment, calls, allocation expressions, or sequences. Constructors are not modeled in the semantics; instead, a `new` expression may explicitly initialize fields of the new object.

A.1 Subclassing and further binding

Subclassing among classes is defined in Figure 9. The judgment $\vdash P \sqsubset_{\mathsf{sc}} P'$ states that P is a declared subclass of P'. The rules SC-OUTER and SC-NEST simply lookup the superclass using the class table CT, substituting the container for `this.class`. The class table, CT, defined in Figure 9 maps class names P to class declarations. We write $CT(P) = \bot$ if P has no definition. SC-INH defines subclassing for implicit classes not in the domain of the class table. The mem function returns the set of classes P comprising a non-dependent type S; a type S is equivalent to the intersection of all classes in $\mathsf{mem}(S)$.

The judgment $\vdash P \sqsubset_{\mathsf{fb}} P'$ states that $P.C$ further binds $P'.C$ when P inherits from P' and $P'.C$ is defined. The judgment $\vdash S\ \mathsf{defined}$ holds when either S is a class in the class table, further binds a defined class, or all its members are defined. We write $\vdash P \sqsubset P'$ if P either subclasses or further binds P'; $\sqsubset^*$ is the reflexive transitive closure of $\sqsubset$. The function $\mathsf{super}(S)$ returns the set of all superclasses of S, including S itself.

A.2 Prefix types

Unlike with virtual classes [16], it is possible in JX/MI to extend classes nested within other namespaces. Multiple nested classes or a mix of top-level and nested classes may be extended, resulting in an intersection of several types with different containers. This feature complicates the definition of prefix types. Consider this example:

```
class A { class B { B x; } }
class A1 extends A { class B { B y = x; } }
class A2 extends A { class B { } }
class C extends A1.B, A2.B { }
```

The name `B` in `A.B` is sugar for the type `A[this.class].B`. The same name `B` in `A1.B` is sugar for `A1[this.class].B`. The two references to `B` should resolve to the same type, thus permitting the assignment from `x` to `y` in `A1.B`; that is, it must be that `A[this.class]` is equivalent to `A1[this.class]`. Similarly, the three types `A[C]`, `A1[C]`, and `A2[C]` should all be equivalent.

The meaning of non-dependent prefix types P`[`S`]` is defined by the prefix function in Figure 9. The P-prefix of a non-dependent type S is the intersection of all classes P' where $P'.C$ is a superclass of S and if P and P' share a common superclass.

A.3 Member lookup

Method and field lookup functions are defined in Figure 10. For a class P, we define $\mathsf{ownFields}(P)$ and $\mathsf{ownMethods}(P)$ to be the set of fields and methods declared in the class. To implement method dispatch ordering, $\mathsf{ownMethods}$ also returns with each method declaration the class in which the method was declared.

Using these definitions, the set of fields and methods declared or inherited by a non-dependent type S is defined by $\mathsf{fields}(S)$ and $\mathsf{methods}(S)$ functions. The ftype function returns the declared type of a field f of an arbitrary type T in environment Γ. The mtype function provides similar. functionality for methods.

The most specific method body for a method m in type S is returned by mbody. Method dispatch order is defined by the $\preceq$ relation. If P_1 inherits from P_2, then P_1's implementation has priority. Further bound classes $P.C$ are ordered before declared superclasses.

A.4 Type well-formedness and simple bounds

The judgment $\Gamma \vdash T \rhd S$ in Figure 11 states that T has a non-dependent bounding type S. For dependent classes p`.class`, the bounding type is simply the bound on the declared type of p. For prefix types P`[`T`]`, the bound is the result of computing the prefix function for P and the bounding type of T.

A type T is well-formed in a context Γ, written $\Gamma \vdash T$, if it has a non-dependent bound and if T's bound is the empty intersection type only if T is the empty intersection.

A.5 Typing

The judgment $\Gamma \vdash_{\mathsf{fin}} p : T$ in Figure 11 states that the access path p is a well-typed final access path in context Γ.

For arbitrary expressions, the judgment $\Gamma \vdash e : T$. states that e has type T in context Γ. The rules for `null`, locations, and variables are standard.

The type of a field access $e_0.f$ is obtained by looking up the the field in T_0, the static type of e_0, then substituting in T_0 for `this` in the field type. Type substitution is defined in Figure 12. Substitution need not preserve exactness of the field type. In contrast, the rule T-SET does require that exactness be preserved when substituting

$\boxed{\vdash S \text{ defined}}$

$$\frac{CT(P) \neq \bot}{\vdash P \text{ defined}} \quad \text{(DEF-EXPL)}$$

$$\frac{\vdash P \sqsubset P' \quad \vdash P'.C \text{ defined}}{\vdash P.C \text{ defined}} \quad \text{(DEF-INH)}$$

$$\frac{\forall P \in \mathsf{mem}(S).\ \vdash P \text{ defined}}{\vdash S \text{ defined}} \quad \text{(DEF-MEM)}$$

$\boxed{\vdash P \sqsubset_{\mathsf{sc}} P'}$

$$\frac{\begin{array}{c} CT(C) = \texttt{class}\ C\ \texttt{extends}\ S\ \{\dots\} \\ P \in \mathsf{mem}(S) \end{array}}{\vdash C \sqsubset_{\mathsf{sc}} P} \quad \text{(SC-OUTER)}$$

$$\frac{\begin{array}{c} CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T\ \{\dots\} \\ \mathsf{substForThis}(T,P) = S \\ P' \in \mathsf{mem}(S) \end{array}}{\vdash P.C \sqsubset_{\mathsf{sc}} P'} \quad \text{(SC-NEST)}$$

$$\frac{\vdash P \sqsubset P' \quad \vdash P.C \sqsubset_{\mathsf{sc}} P.C'}{\vdash P.C \sqsubset_{\mathsf{sc}} P.C'} \quad \text{(SC-INH)}$$

$\boxed{\vdash P \sqsubset_{\mathsf{fb}} P'}$

$$\frac{\vdash P \sqsubset P' \quad \vdash P'.C \text{ defined}}{\vdash P.C \sqsubset_{\mathsf{fb}} P'.C} \quad \text{(FB)}$$

$\boxed{\vdash P \sqsubset P'}$

$$\frac{\vdash P \sqsubset_{\mathsf{sc}} P'}{\vdash P \sqsubset P'} \quad \text{(INH-SC)}$$

$$\frac{\vdash P \sqsubset_{\mathsf{fb}} P'}{\vdash P \sqsubset P'} \quad \text{(INH-FB)}$$

$$\frac{\begin{array}{c} Pr = \langle \overline{L}, e \rangle \\ \texttt{class}\ C\ \texttt{extends}\ T\ \{\dots\} \in \overline{L} \end{array}}{CT(C) = \texttt{class}\ C\ \texttt{extends}\ T\ \{\dots\}}$$

$$\frac{\begin{array}{c} CT(P) = \texttt{class}\ C'\ \texttt{extends}\ T'\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \texttt{class}\ C\ \texttt{extends}\ T\ \{\dots\} \in \overline{L'} \end{array}}{CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T\ \{\dots\}}$$

$$\begin{aligned} \mathsf{substForThis}(C, T') &= C \\ \mathsf{substForThis}(T.C, T') &= \mathsf{substForThis}(T, T').C \\ \mathsf{substForThis}(P\texttt{[}T\texttt{]}, T') &= P\texttt{[}\mathsf{substForThis}(T, T')\texttt{]} \\ \mathsf{substForThis}(\texttt{this.class}, T') &= T' \\ \mathsf{substForThis}(\&\overline{T}, T') &= \&(\mathsf{substForThis}(\overline{T}, T')) \end{aligned}$$

$$\mathsf{mem}(P) = \{P\}$$

$$\frac{\mathsf{mem}(S) = \{\overline{S}\}}{\mathsf{mem}(S.C) = \{\overline{S}.C\}}$$

$$\frac{\mathsf{prefix}(P,S) = S'}{\mathsf{mem}(P\texttt{[}S\texttt{]}) = \mathsf{mem}(S')}$$

$$\mathsf{mem}(\&\overline{S}) = \bigcup_{S_i \in \overline{S}} \mathsf{mem}(S_i)$$

$$\mathsf{supers}(S) = \bigcup_{P \in \mathsf{mem}(S)} \{P' \mid \vdash P \sqsubset^* P'\}$$

$$\begin{aligned} \mathsf{prefix}(P,S) = \&\{P' \mid \exists C.\ & P'.C \in \mathsf{supers}(S) \\ & \wedge \big(\mathsf{supers}(P) \cap \mathsf{supers}(P')\big) \neq \emptyset\} \end{aligned}$$

Figure 9. Subclassing and auxiliary functions

the field target type into the field type. This ensures that the if the field type is dependent on `this`, the value assigned into the field has an appropriate type dependent on the target type.

Calls are checked by looking up the method type, then substituting in the receiver type and the actual argument types for `this` and the formal parameters. The actuals must have the same type as the substituted formal types, preserving exactness for the same reason as in T-SET. Subtitution of the return type need not preserve exactness.

A `new` T expression is well-typed if all it initializes only declared fields of a well-formed type T. A sequence expression takes the type of the second expression in the sequence.

Finally, the rule T-FIN allows a final access path p to be coerced to type p.`class`, and T-SUB is the standard subsumption rule.

A.6 Subtyping and type equivalence

Subtyping rules are defined in Figure 11. The judgment $\Gamma \vdash T \leq T'$ states that T is a subtype of T' in context Γ. The rules ensure that syntactically different types representing the same sets of values are considered equal. The judgment $\Gamma \vdash T \approx T'$ is sugar for the pair of judgments $\Gamma \vdash T \leq T'$ and $\Gamma \vdash T' \leq T$.

Subtyping is reflexive and transitive. The rules S-SUP state that a type is a subclass of its declared superclass; the enclosing class of the subtype T is substituted in for `this` in the superclass.

The rule S-NEST states that a nested class C is covariant with its containing class; that is, further binding implies subtyping. S-BOUND states that a type is a subtype of its bounding simple type. Since all types contain the `null` value, `null.class` is a subtype of any well-formed type.

S-OUT, S-IN, and S-PRE relate prefix types to non-prefix types. S-PRE extends the definition of prefix to arbitrary types.

In JX/MI, unlike with virtual classes [16], it is possible to extend classes nested within other namespaces. Multiple nested classes or a mix of top-level and nested classes may be extended, resulting in an intersection of several types with different containers. This feature complicates equality of prefix types. Consider this example:

```
class A { class B { B x; } }
class A1 extends A { class B { B y = x; } }
class A2 extends A { class B { } }
class C extends A1.B, A2.B { }
```

The name `B` in `A.B` is interpreted as the type `A[this.class].B`. When inherited into `A1`, the name should resolve to an equivalent type, thus permitting the assignment from `x` to `y` in `A1.B`. In `A1.B`, the name `B` is interpreted as the type `A1[this.class].B`. Therefore, it must be that `A[this.class]` is equivalent to `A1[this.class]`. Similarly, `A[C]`, `A1[C]`, and `A2[C]` are all equivalent to each other. The rule S-PRE ensures this equivalence relation.

S-MEET-LB and S-MEET-G are from Compagnoni and Pierce [10] and define subtyping for intersection types. Together these two rules imply that intersection types are associative and

$$\frac{CT(P) = \texttt{class}\ C\ \texttt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}}{\begin{array}{c}\mathsf{ownFields}(P) = \overline{F}\\ \mathsf{ownMethods}(P) = \{(P,M_i) \mid M_i \in \overline{M}\}\end{array}}$$

$$\frac{CT(P) = \bot}{\begin{array}{c}\mathsf{ownFields}(P) = \emptyset\\ \mathsf{ownMethods}(P) = \emptyset\end{array}}$$

$$\mathsf{fields}(S) = \bigcup_{P_i \in \overline{P}} \mathsf{ownFields}(P_i)$$

$$\mathsf{methods}(S) = \bigcup_{P_i \in \overline{P}} \mathsf{ownMethods}(P_i)$$

where $\overline{P} = \{P' \mid P \in \mathsf{mem}(S) \wedge \vdash P \sqsubset^* P'\}$.

$$\frac{\begin{array}{c}\Gamma \vdash T \rhd S \quad \mathsf{fields}(S) = \overline{F}\\ F_i = [\texttt{final}]\ T_f\ f = e\end{array}}{\mathsf{ftype}(\Gamma, T, f) = T_f}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \rhd S \quad \mathsf{methods}(S) = \overline{(P,M)}\\ M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \quad M \in \overline{M}\end{array}}{\mathsf{mtype}(\Gamma, T, m) = (\overline{x}\!:\!\overline{T}) \rightarrow T_{n+1}}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \rhd S \quad \mathsf{methods}(S) = \overline{(P,M)}\\ M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\\ M_i = \mathsf{mostSpecific}(m, S, \overline{(P,M)})\end{array}}{\mathsf{mbody}(S, m) = M_i}$$

$$\frac{\vdash P_1 \sqsubset^* P_2}{S \vdash P_1 \preceq P_2}$$

$$\frac{P \in \mathsf{mem}(S) \quad \vdash P \sqsubset_{\mathsf{fb}} P_1 \quad \vdash P \sqsubset_{\mathsf{sc}} P_2}{S \vdash P_1 \preceq P_2}$$

$$\frac{\begin{array}{c}M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\\ \forall j.\ \left(\begin{array}{l}M_j = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\}\\ \Rightarrow S \vdash P_i \preceq P_j\end{array}\right)\end{array}}{M_i = \mathsf{mostSpecific}(m, S, \overline{(P,M)})}$$

Figure 10. Lookup functions

$\boxed{\Gamma \vdash T \rhd S}$

$$\frac{\forall i.\ \Gamma \vdash T_i \rhd S_i}{\Gamma \vdash \&\overline{T} \rhd \&\overline{S}} \qquad \frac{CT(P) \neq \bot}{\Gamma \vdash P \rhd P} \qquad \frac{\begin{array}{c}\Gamma \vdash T \rhd S\\ \vdash S.C \text{ defined}\end{array}}{\Gamma \vdash T.C \rhd S.C} \qquad \frac{\Gamma \vdash_{\mathsf{fin}} p : T \quad \Gamma \vdash T \rhd S}{\Gamma \vdash p.\texttt{class} \rhd S} \qquad \frac{\Gamma \vdash T \rhd S \quad \mathsf{prefix}(P,S) = S'}{\Gamma \vdash P\texttt{[}T\texttt{]} \rhd S'}$$

$\boxed{\Gamma \vdash T}$

$$\frac{\Gamma \vdash T \rhd S \quad T \neq \&\emptyset \Rightarrow S \neq \&\emptyset}{\Gamma \vdash T}$$

$\boxed{\Gamma \vdash_{\mathsf{fin}} p : T}$

$$\frac{\Gamma \vdash T}{\Gamma \vdash_{\mathsf{fin}} \texttt{null} : T}\ \text{(F-NULL)} \qquad \frac{\Gamma \vdash S}{\Gamma \vdash_{\mathsf{fin}} \ell_S : S}\ \text{(F-LOC)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash_{\mathsf{fin}} x : T}\ \text{(F-VAR)} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{fin}} p : T\\ \mathsf{ftype}(\Gamma, T, f) = \texttt{final}\ T_f\\ T_f\{p/\texttt{this}\} = T'_f\end{array}}{\Gamma \vdash_{\mathsf{fin}} p.f : T'_f}\ \text{(F-GET)}$$

$\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash T}{\Gamma \vdash \texttt{null} : T}\ \text{(T-NULL)} \qquad \frac{\Gamma \vdash S}{\Gamma \vdash \ell_S : S}\ \text{(T-LOC)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ \text{(T-VAR)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : T_0\\ \mathsf{ftype}(\Gamma, T_0, f) = [\texttt{final}]\ T_f\\ \Gamma \vdash T_f\{\!\!\{T/\texttt{this}\}\!\!\} = T'_f\end{array}}{\Gamma \vdash e_0.f : T'_f}\ \text{(T-GET)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \quad \Gamma \vdash e_1 : T'_f\\ \mathsf{ftype}(\Gamma, T_0, f) = T_f\\ \Gamma \vdash T_f\{\!\!\{T_0/\texttt{this}\}\!\!\}_{\times} = T'_f\end{array}}{\Gamma \vdash e_0.f = e_1 : T'_f}\ \text{(T-SET)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0\\ \mathsf{mtype}(\Gamma, T_0, m) = (\overline{x}\!:\!\overline{T}) \rightarrow T_{n+1}\\ n = \#(\overline{e}) = \#(\overline{x}) \quad \Gamma \vdash \overline{e} : \overline{T'}\\ \forall T_i \in \overline{T}.\ \Gamma \vdash T_i\{\!\!\{T_0, \overline{T'}/\texttt{this}, \overline{x}\}\!\!\}_{\times} = T'_i\\ \Gamma \vdash T_{n+1}\{\!\!\{T_0, \overline{T'}/\texttt{this}, \overline{x}\}\!\!\} = T'_{n+1}\end{array}}{\Gamma \vdash e_0.m(\overline{e}) : T'_{n+1}}\ \text{(T-CALL)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \quad \Gamma \vdash \overline{e} : \overline{T}\\ \forall i.\ \mathsf{ftype}(\Gamma, T, f_i) = [\texttt{final}]\ T_i\end{array}}{\Gamma \vdash \texttt{new}\ T(\overline{f} = \overline{e}) : T}\ \text{(T-NEW)} \qquad \frac{\Gamma \vdash e_1 \vdash T_1 \quad \Gamma \vdash e_2 \vdash T_2}{\Gamma \vdash e_1;\ e_2 : T_2}\ \text{(T-SEQ)}$$

$$\frac{\Gamma \vdash_{\mathsf{fin}} p : T}{\Gamma \vdash p : p.\texttt{class}}\ \text{(T-FIN)} \qquad \frac{\Gamma \vdash e : T \quad \Gamma \vdash T \leq T'}{\Gamma \vdash e : T'}\ \text{(T-SUB)}$$

$\boxed{\Gamma \vdash T \leq T'}$

$$\Gamma \vdash T \leq T\ \text{(S-REFL)} \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 \leq T_2\\ \Gamma \vdash T_2 \leq T_3\end{array}}{\Gamma \vdash T_1 \leq T_3}\ \text{(S-TRANS)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \leq P\\ CT(P) = \texttt{class}\ C\ \texttt{extends}\ T'\ \{\ldots\}\\ P = P'.C \Rightarrow \mathsf{substForThis}(T', P'\texttt{[}T\texttt{]}) = T''\\ P = C \Rightarrow T' = T''\end{array}}{\Gamma \vdash T \leq T''}\ \text{(S-SUP)} \qquad \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T'.C}{\Gamma \vdash T.C \leq T'.C}\ \text{(S-NEST)}$$

$$\frac{\Gamma \vdash T \rhd S}{\Gamma \vdash T \leq S}\ \text{(S-BOUND)} \qquad \frac{\Gamma \vdash T}{\Gamma \vdash \texttt{null.class} \leq T}\ \text{(S-NULL)} \qquad \frac{\Gamma \vdash T \leq P.C \quad \Gamma \vdash P\texttt{[}T\texttt{]}}{\Gamma \vdash T \leq P\texttt{[}T\texttt{]}.C}\ \text{(S-OUT)} \qquad \frac{\Gamma \vdash T \leq P \quad \Gamma \vdash T.C}{\Gamma \vdash T \approx P\texttt{[}T.C\texttt{]}}\ \text{(S-IN)}$$

$$\frac{\Gamma \vdash T \leq T'.C \quad \vdash P \sqsubset^* P' \quad \Gamma \vdash T' \leq P'}{\Gamma \vdash P\texttt{[}T\texttt{]} \leq T'}\ \text{(S-PRE)} \qquad \Gamma \vdash \&\overline{T} \leq T_i\ \text{(S-MEET-LB)} \qquad \frac{\forall i.\ \Gamma \vdash T \leq T_i}{\Gamma \vdash T \leq \&\overline{T}}\ \text{(S-MEET-G)}$$

$$\frac{\mathsf{mem}(S_1) = \mathsf{mem}(S_2)}{\Gamma \vdash \ell_{S_1}.\texttt{class} \approx \ell'_{S_2}.\texttt{class}}\ \text{(S-LOC-LOC)} \qquad \frac{\begin{array}{c}\Gamma \vdash T \approx \ell'_{S'}.\texttt{class}\\ \mathsf{mem}(S) = \mathsf{mem}(\mathsf{prefix}(P, S'))\end{array}}{\Gamma \vdash \ell_S.\texttt{class} \approx P\texttt{[}T\texttt{]}}\ \text{(S-LOC-PRE)}$$

Figure 11. Static semantics

$\boxed{\Gamma \vdash T\{\!\{T_s/x\}\!\} = T'}$

$$\frac{\forall i.\ \Gamma \vdash T_i\{\!\{T_s/x\}\!\} = T_i'}{\Gamma \vdash \&\overline{T}\{\!\{T_s/x\}\!\} = \&\overline{T'}}$$

$$\Gamma \vdash C\{\!\{T_s/x\}\!\} = C$$

$$\frac{\Gamma \vdash T\{\!\{T_s/x\}\!\}_{\mathsf{x}} = T'}{\Gamma \vdash T.C\{\!\{T_s/x\}\!\} = T'.C}$$

$$\Gamma \vdash v.\mathtt{class}\{\!\{T_s/x\}\!\} = v.\mathtt{class}$$

$$\frac{x \neq y}{\Gamma \vdash y.\mathtt{class}\{\!\{T_s/x\}\!\} = y.\mathtt{class}}$$

$$\Gamma \vdash x.\mathtt{class}\{\!\{T_s/x\}\!\} = T_s$$

$$\frac{\begin{array}{c}\mathsf{ftype}(\Gamma, p.\mathtt{class}, f) = T_f \\ \Gamma \vdash T_f\{p/\mathtt{this}\}\{\!\{T_s/x\}\!\} = T_f''\end{array}}{\Gamma \vdash p.f.\mathtt{class}\{\!\{T_s/x\}\!\} = T_f''}$$

$$\frac{\Gamma \vdash p.\mathtt{class}\{\!\{p'.\mathtt{class}/x\}\!\}_{\mathsf{x}} = p''.\mathtt{class}}{\Gamma \vdash p.f.\mathtt{class}\{\!\{p'.\mathtt{class}/x\}\!\} = p''.f.\mathtt{class}}$$

$$\frac{\Gamma \vdash T\{\!\{T_s/x\}\!\} = T'}{\Gamma \vdash P[T]\{\!\{T_s/x\}\!\} = P[T']}$$

$\boxed{\Gamma \vdash T\{\!\{T_s/x\}\!\}_{\mathsf{x}} = T'}$

$$\frac{\Gamma \vdash T\{\!\{T_s/x\}\!\} = T' \quad \mathsf{exact}(T) \Rightarrow \mathsf{exact}(T')}{\Gamma \vdash T\{\!\{T_s/x\}\!\}_{\mathsf{x}} = T'}$$

Figure 12. Type substitution

commutative and that the singleton intersection type $\&T$ is equivalent to its element type T.

The rules are expressive enough to derive the judgments $\Gamma \vdash P[\overline{T}] \leq P[T_i]$ and $\Gamma \vdash \overline{T}.C \leq T_i.C$.

Two locations that point to objects of the same simple type are equal by S-LOC-LOC; S-LOC-PRE similarly equates a location-dependent class and a prefix type.

A.7 Program typing

Program typing rules are presented in Figure 14. The program Pr is well-formed if all class declarations are well-formed and if the "main" expression is well-typed.

A class declaration is well-formed if all its members are well-formed and its superclass is well-formed in an environment containing only `this`. Additionally, the class must conform to all of its superclasses.

A.8 Operational semantics

A small-step operational semantics is shown in Figure 15. The semantics are defined using a reduction relation $\longrightarrow$, which maps a configuration of an expression e and a heap H to a new configuration. A heap H is a function from memory locations ℓ_S to objects $S\ \{\overline{f} = \overline{v}\}$. The notation $e, H \longrightarrow r, H'$ means that expression e and heap H step to result r and heap H'. Results are either expressions

$\boxed{\vdash \Gamma\ \mathsf{ok}}$

$$\vdash \emptyset\ \mathsf{ok}$$

$$\frac{\vdash \Gamma\ \mathsf{ok} \quad x \notin \mathrm{dom}(\Gamma) \quad \Gamma \vdash T}{\vdash \Gamma, x\!:\!T\ \mathsf{ok}}$$

$$\frac{\vdash \Gamma'\ \mathsf{ok} \quad \Gamma \text{ permutes } \Gamma'}{\vdash \Gamma\ \mathsf{ok}}$$

Figure 13. Well-formed environments

$$\frac{\vdash \overline{L}\ \mathsf{ok} \quad \emptyset \vdash e\!:\!T \quad \emptyset \vdash T \quad \sqsubset^+ \text{ acyclic}}{\vdash \langle \overline{L}, e\rangle\ \mathsf{ok}} \quad \text{(P-OK)}$$

$$\frac{\begin{array}{c} P.C \vdash \overline{L}\ \mathsf{ok} \quad P.C \vdash \overline{F}\ \mathsf{ok} \quad P.C \vdash \overline{M}\ \mathsf{ok} \\ (P \neq \mathsf{nil} \Rightarrow \mathtt{this}\!:\!P \vdash T\ \mathsf{ok}) \quad (P = \mathsf{nil} \Rightarrow \emptyset \vdash T\ \mathsf{ok}) \\ \mathsf{supers}(P.C) = \overline{P} \quad \forall i.\ P \vdash P.C \text{ conforms to } P_i \end{array}}{P \vdash \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}\ \mathsf{ok}} \quad \text{(L-OK)}$$

$$\frac{\begin{array}{c} CT(P.C) = \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ CT(P') = \mathtt{class}\ C'\ \mathtt{extends}\ T'\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \forall D \in \mathrm{dom}(\overline{L'} \cap \overline{L}).\ P.C \vdash P.C.D \text{ extends super of } P'.D \\ M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \quad M \in \overline{M} \\ M' = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\} \quad M' \in \overline{M'} \\ P.C \vdash M \text{ overrides method } M' \end{array}}{P \vdash P.C \text{ conforms to } P'}$$

$$\frac{\begin{array}{c} M = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \\ M' = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\} \\ \#(\overline{x}) = \#(\overline{x'}) = \#(\overline{y}) \quad \overline{y} \cap (\overline{x} \cup \overline{x'}) = \emptyset \\ \Gamma = \mathtt{this}\!:\!P, \overline{y}\!:\!\overline{T}\{\overline{y}/\overline{x}\} \\ \vdash \Gamma\ \mathsf{ok} \\ \Gamma \vdash \overline{T'}\{\overline{y}/\overline{x'}\} \leq \overline{T}\{\overline{y}/\overline{x}\} \\ \Gamma \vdash T_{n+1}\{\overline{y}/\overline{x}\} \leq T'_{n+1}\{\overline{y}/\overline{x'}\} \end{array}}{P \vdash M \text{ overrides method } M'}$$

$$\frac{\begin{array}{c} CT(P.C) = \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\ldots\} \\ CT(P') = \mathtt{class}\ C'\ \mathtt{extends}\ T'\ \{\ldots\} \\ \mathsf{substForThis}(T, P) = S \quad \mathsf{substForThis}(T', P) = S' \\ \vdash S \sqsubset^* S' \end{array}}{P \vdash P.C \text{ extends super of } P'}$$

$$\frac{\mathtt{this}\!:\!P \vdash T\ \mathsf{ok} \quad \emptyset \vdash e\!:\!T}{P \vdash \mathtt{final}\ T\ f = e\ \mathsf{ok}} \quad \text{(F-OK)}$$

$$\frac{\mathtt{this}\!:\!P \vdash T\ \mathsf{ok} \quad \vdash \mathtt{this}\!:\!P, \overline{x}\!:\!\overline{T}\ \mathsf{ok} \quad \mathtt{this}\!:\!P, \overline{x}\!:\!\overline{T} \vdash e\!:\!T}{P \vdash T\ m(\overline{T}\ \overline{x})\ \{e\}\ \mathsf{ok}} \quad \text{(M-OK)}$$

Figure 14. Program typing

$$\boxed{e, H \longrightarrow r, H}$$

objects	o	$::= S\ \{\overline{f} = \overline{v}\}$
results	r	$::= e \mid \mathsf{NullError}$
evaluation contexts	E	$::= [\cdot]$
		$\mid E.f$
		$\mid \texttt{new}\ TE(\overline{f} = \overline{e})$
		$\mid \texttt{new}\ S(\overline{f} = \overline{v}, f = E, \overline{f'} = \overline{e'})$
		$\mid E.f = e$
		$\mid \ell_S.f = E$
		$\mid E.m(\overline{e})$
		$\mid \ell_S.m(\overline{v}, E, \overline{e'})$
		$\mid E;\ e$
type evaluation contexts	TE	$::= \&(\overline{S}, TE, \overline{T'})$
		$\mid TE.C$
		$\mid E.\texttt{class}$
		$\mid P[TE]$
null error contexts	NE	$::= \texttt{null}.f$
		$\mid \texttt{null}.f = e$
		$\mid \texttt{null}.m(\overline{e})$
		$\mid \texttt{new}\ TE[\texttt{null}](\overline{f} = \overline{e})$

$$\frac{e, H \longrightarrow e', H'}{E[e], H \longrightarrow E[e'], H'} \qquad \text{(R-CONG)}$$

$$E[NE], H \longrightarrow \mathsf{NullError}, H \qquad \text{(R-NULL)}$$

$$\frac{H(\ell_S) = S\ \{\overline{f} = \overline{v}\}}{\ell_S.f_i, H \longrightarrow v'_i, H} \qquad \text{(R-GET)}$$

$$\frac{\begin{array}{c} H(\ell_S) = S\ \{\overline{f} = \overline{v}\} \\ H' = H[\ell_S := S\ \{f_1 = v_1, \ldots, f_i = v, \ldots, f_n = v_n\}] \end{array}}{\ell_S.f = v, H \longrightarrow v, H'} \qquad \text{(R-SET)}$$

$$\frac{\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \quad \#(\overline{v}) = \#(\overline{x})}{\ell_S.m(\overline{v}), H \longrightarrow e\{\ell_S/\texttt{this}, \overline{v}/\overline{x}\}, H} \qquad \text{(R-CALL)}$$

$$\frac{\begin{array}{c} \vdash T \rhd S \quad \mathsf{fields}(S) = \overline{F}, \overline{F'} \\ \overline{F} = [\texttt{final}]\ \overline{T}\ \overline{f} = \overline{e} \quad \overline{F'} = [\texttt{final}]\ \overline{T'}\ \overline{f'} = \overline{e'} \end{array}}{\texttt{new}\ T(\overline{f} = \overline{v}), H \longrightarrow \texttt{new}\ S(\overline{f} = \overline{v}, \overline{f'} = \overline{e'}), H} \qquad \text{(R-NEW)}$$

$$\frac{\begin{array}{c} \mathsf{fields}(S) = \overline{F} \quad \#(\overline{f}) = \#(\bar{(F)}) \\ \ell_S \notin \mathrm{dom}(H) \quad H' = H[\ell_S := S\ \{\overline{f} = \overline{v}\}] \end{array}}{\texttt{new}\ S(\overline{f} = \overline{v}), H \longrightarrow \ell_S, H'} \qquad \text{(R-ALLOC)}$$

$$v;\ e, H \longrightarrow e, H \qquad \text{(R-SEQ)}$$

Figure 15. Operational semantics

or NullError The initial configuration for program $\langle \overline{L}, e\rangle$ is $e, \emptyset$. Final configurations are of the form v, H, or $\mathsf{NullError}, H$.

The reduction rules are mostly straightforward. Order of evaluation is captured by an evaluation context E (an expression with a hole $[\cdot]$) and the congruence rule R-CONG. The rule R-NULL propagates a dereference of a `null` pointer out through the evaluation contexts to produce a NullError, simulating a Java `NullPointerException`.

R-GET and R-SET get or set a field in a heap object, respectively. R-CALL uses the mbody function defined in Figure 10 to locate the most specific implementation of method m.

There are two rules for evaluating `new` expressions. R-NEW looks up all fields of the type being allocated and steps to a configuration containing initializers for those fields. R-ALLOC is applied when all initializers have been evaluated. A new location is allocated and the object is installed in the heap.

A.9 Heap typing

Figure 16 shows the heap typing rules. A heap H is well-formed if all locations in its domains refer to objects of the appropriate type. A configuration (e, H) is well-formed if all H is well-formed and all free locations in e are in H.

A.10 Soundness

To prove soundness we use the standard technique of proving subjuct reduction and progress lemmas [46]. The key lemmas are stated here.

The subject reduction lemma states that a well-formed configuration steps to another well-formed configuration or to a configuration containing NullError.

LEMMA A.1. (Subject reduction) *If $\emptyset \vdash e : T$, $\vdash e, H$, and $e, H \longrightarrow r, H'$, then $\vdash \langle r, H'\rangle$ and either $r = e'$ and $\emptyset \vdash e' : T$ or $r = \mathsf{NullError}$.*

$$\frac{\vdash H \quad \forall \ell_S \in \mathsf{locs}(e).\ \ell_S \in \mathrm{dom}(H)}{\vdash e, H} \qquad \text{(CONFIG)}$$

$$\frac{\begin{array}{c} \mathsf{fields}(S) = [\texttt{final}]\ \overline{T}\ \overline{f} = \overline{e} \\ H(\ell_S) = S\ \{\overline{f} = \overline{v}\} \quad \emptyset \vdash \overline{v} : \overline{T}\{\ell_S/\texttt{this}\} \quad \vdash \overline{v}, H \quad H \vdash \overline{v} \end{array}}{H \vdash \ell_S} \qquad \text{(H-LOC)}$$

$$H \vdash \texttt{null} \qquad \text{(H-NULL)}$$

$$\frac{\forall \ell_S \in \mathrm{dom}(H).\ H \vdash \ell_S}{\vdash H} \qquad \text{(HEAP)}$$

Figure 16. Heap typing

The progress lemma states that for any well-formed configuration e, H, either e is a value or e, H steps to a new configuration r, H'.

LEMMA A.2. (Progress) *If $\emptyset \vdash e : T$, $\vdash e, H$, then either $e = v$, or there is a H' and an r such that $e, H \longrightarrow r, H'$.*

Soundness follows directly from the subject reduction and progress lemma.

THEOREM A.3. (Soundness) *If $\vdash \langle \overline{L}, e\rangle$ ok and $\vdash e : T$, then there is an r such that $r = v$ and $\vdash v : T$ or $r = \mathsf{NullError}$, and $e, \emptyset \longrightarrow^* r, H'$.*