J&: Nested Intersection for Scalable Software Composition

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 compose large software systems, combining their features in a modular way. J& supports *nested intersection*, 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. Nested intersection enables composition and extension of *two or more* packages, combining their types and behavior while resolving conflicts with a relatively small amount of code. The utility of J& 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

Most software is constructed by extending and composing existing code. 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 contain many mutually dependent classes, functions, and types. Moreover, these mechanisms do not adequately support *composition* of multiple interacting classes. Better language support is needed.

This paper introduces the language J& (pronounced "Jet"), which supports the scalable, modular composition and extension of large software frameworks. J& builds on the Java-based language Jx, which supports scalable extension of software frameworks through *nested inheritance* [32]. J& adds a new language feature, *nested intersection*, which enables composition of multiple software frameworks to obtain a software system that combines their functionality.

Programmers are familiar with a simple form of software composition: linking, which works when the composed software components offer disjoint, complementary functionality. In the general case, two software components are not disjoint. They may in fact offer similar functionality, because they extend a common ancestor component. Composing related frameworks should integrate their extensions rather than duplicating the extended components. It is this more general form of software composition that nested intersection supports.

A motivating example for software composition was the problem of combining domain-specific compiler extensions. We demonstrate the utility of nested intersection through a J& compiler framework for implementing domain-specific extensions to the Java language. Using the framework, which is based on the Polyglot compiler framework [33], one can choose useful language features for a given application domain from a "menu" of available options, then compose the corresponding compilers to obtain a compiler for the desired language.

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

1. Orthogonal extension: Extensions may require both new data types and new operations.
2. Type safety: Extensions cannot create run-time type errors.
3. Modularity: The base system can be extended without modifying or recompiling its code.
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* [44]. Scalability (4) is often but not necessarily satisfied by supporting separate compilation; it is important for extending large software. Non-destructive extension (5) 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. Nested inheritance [32] addresses the first five requirements, but it does not support extension composition. Nested intersection adds this capability.

This paper describes nested intersection in the J& 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 nested intersection and J&. Nested intersection creates several interesting technical challenges, such as the problem of resolving conflicts among composed packages; this topic and a detailed discussion of language semantics are presented in Section 3. Section 4 then describes how nested intersection is used to extend and compose compilers. The implementation of J& is described in Section 5, and Section 6 describes experience using J& to implement and compose extensions in the Polyglot compiler framework and in the Pastry framework for building peer-to-peer systems [39]. Related work is discussed in Section 7, and the paper concludes in Section 8. The appendix gives a formal operational semantics and a type system for J&.

2. Nested intersection

Nested intersection supports scalable extension of a base system and scalable composition of those extensions. To illustrate how nested intersection achieves this goal, we consider the example of building a compiler with composable extensions. A compiler

```
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;
  Exp accept(Visitor v) {
    fst.accept(v); snd.accept(v);
    return v.visitPair(this);
  }
}
class Visitor {
  Exp visitPair(Pair p) { return p; }
}
class TypeChecker extends Visitor {
  Exp visitPair(Pair 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());
  }
}
```

(a) Lambda calculus + pairs compilers

```
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() { ... }
}
```

(b) Lambda calculus + sums compiler

```
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());
  }
}
```

(c) Conflict resolution

Figure 1. Compiler composition

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: syntax, types, analyses, optimizations.

2.1 Nested inheritance

Nested intersection builds on previous work on nested inheritance [32], 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; J& extends Jx with nested intersection.

Figure 1(a) 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.

Nested inheritance in J& is inheritance of *namespaces*, that is, packages and classes. 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.

As with virtual classes [27, 28, 18], overriding of a nested class does not replace the original class, but instead refines, or *further binds* [27], it. If a namespace T' extends another namespace T that contains a nested namespace $T.C$, then $T'.C$ inherits members from $T.C$ as well as from $T'.C$'s explicitly named base namespaces (if any). Further binding thus provides a limited form of multiple inheritance: *explicit inheritance* from the named base of $T'.C$ and *induced inheritance* from the original namespace $T.C$. Unlike with virtual classes, $T'.C$ is also a subtype of $T.C$. In Figure 1(a), the `pair` package extends the `base` package, further binding the `Visitor`, `TypeChecker`, and `Compiler` classes, as illustrated by the `base` and `pair` boxes in the inheritance hierarchy of Figure 2. The class `pair.TypeChecker` is a subclass of both `base.TypeChecker` and `pair.Visitor` and contains both the `visitAbs` and `visitPair` methods.

The key feature of nested inheritance that enables scalable extensibility is late binding of type names. When the name of a class or package is inherited into a new namespace, the name is interpreted in the context of the namespace into which it was inherited, rather than where it was originally defined. When the name occurs in a method body, the type it represents may depend on the run-time value of `this`.

In Figure 1(a), the name `Visitor`, in the context of the `base` package, is interpreted as `base.Visitor`. In the context of `pair`, which inherits from `base`, `Visitor` refers to `pair.Visitor`. Thus, when the method `accept` is called on an instance of `base.Abs`, it must be called with a `base.Visitor` and *not* with a `pair.Visitor`. Similarly, the method `pair.Pair.accept` can only be called with a `pair.Visitor`, allowing it to access the parameter's `visitPair` method.

Late binding applies to supertype declarations as well. Thus, `pair.Emitter` extends `pair.Visitor` and inherits its `visitPair` method. Late binding of supertype declarations thus provides a form of *virtual superclasses* [28, 14], permitting inheritance relationships among the nested namespaces to be preserved when inherited into a new enclosing namespace. The class hierar-

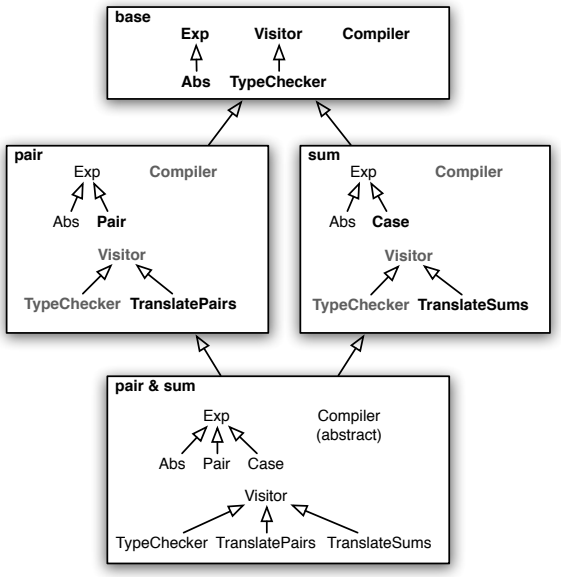

Figure 2. Inheritance hierarchy for compiler composition

chy in the original namespace is replicated in the derived namespace, and in that derived namespace, when a class is further bound, new members added into it are automatically inherited by subclasses in the new hierarchy.

Compilers are constructed of sets of related, mutually dependent classes. For example, the classes `Exp` and `Visitor` in the `base` package 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, the programmer may want some references to other types to be late bound, while others should refer to a particular fixed class. Late binding is achieved by interpreting unqualified type names like `Visitor` as sugar for types nested within *dependent classes* and *prefix types*. The semantics of these types are described in more detail in Section 3. Usually, the programmer need not write down these desugared types; most Jx and J& code looks and behaves like Java code.

2.2 Extensibility requirements

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

Orthogonal extension. Compiler frameworks must support the addition of both new data types (e.g., abstract syntax, types, dataflow analysis values) and operations on those types (e.g., type checking, optimization, translation). 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 late binding of type names causes inherited methods to operate automatically on data types further bound in the inheriting context.

Type safety. Nested inheritance is also type-safe [32]. Dependent classes ensure that extension code cannot use objects of the base system or of other extensions as if they belonged to the extension, which could cause run-time errors.

Modularity and scalability. Extensions are subclasses (or subpackages) and hence are modular. 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.

Non-destructive extension. Nested inheritance does not affect the base code, so it is a non-destructive extension mechanism, unlike open classes [11] and aspects [25]. Therefore, base 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.

The sixth requirement, composition of extensions, is discussed in the next section.

2.3 Composition

To support composition of extensions, J& extends Jx with nested intersection: New classes and packages may be constructed by inheriting from multiple packages or classes; the class hierarchies nested within the base namespaces are composed to achieve a composition of their functionalities.

For two namespaces S and T, $S\,\&\,T$ is the *intersection* of these two namespaces. Nested intersection is a form of multiple inheritance implemented using *intersection types* [38, 12]: $S\,\&\,T$ inherits from both S and T and is a subtype of both S and T.

Nested intersection is most useful when composing related packages or classes. When two namespaces that both extend a common base namespace are intersected, their common nested namespaces are themselves intersected: if S and T contain nested namespaces $S.C$ and $T.C$, the intersection $S\,\&\,T$ contains $(S\,\&\,T).C$, which is equal to $S.C\,\&\,T.C$.

Consider the lambda calculus compiler from Figure 1(a). Suppose that we had also extended the `base` package to a `sum` package implementing a compiler for the lambda calculus extended with sum types. This compiler is shown in Figures 1(b).

The intersection package `pair & sum`, shown in Figure 2, composes the two compilers, producing a compiler for the lambda calculus extended with both product and sum types. Since both `pair` and `sum` contain a class `Compiler`, the new class `(pair & sum).Compiler` extends both `pair.Compiler` and `sum.Compiler`. However, both `pair.Compiler` and `sum.Compiler` define a method `main`. The class `(pair & sum).Compiler` thus contains conflicting versions of `main`. The conflict is resolved in Figure 1(c) by creating a new derived package `pair_and_sum` that overrides `main`, defining the order of compiler passes for the composed compiler.

3. Semantics of J&

This section gives an overview of the static and dynamic semantics of J&. Formal details as well as a sketch of a proof of soundness are presented in Appendix A. To conserve space, not all the features of the J& language are discussed in detail; package inheritance, in particular, is discussed only briefly.

3.1 Dependent classes and prefix types

In most cases, code in Jx and J& looks and behaves like Java code. However, unqualified type names are really syntactic sugar for dependent classes and prefix types.

The *dependent class* p.`class` 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; or, 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. In general, the class represented by p.`class` is statically unknown, but fixed: for a particular p, all members of p.`class` have the same run-time class as the object referred to by p.

The *prefix type* P`[`T`]` represents the innermost enclosing namespace of T that is a subtype of the namespace P. P must be a non-dependent type: either a top-level namespace C or of the form $P'.C$. J& has both prefix classes and prefix packages. P`[`T`]` may be defined even if T is a top-level namespace because T may be a subtype of a namespace contained in P.

The expressiveness of prefix types makes it possible in Jx and J& to extend a single class that is part of a collection of related classes, creating a subclass in a new container, without extending all classes in the original collection. Inheritance can operate at every level of the containment hierarchy, enabling fine-grained reuse of individual classes in other namespaces. Because a class may be extended by another class in a different containing namespace, the subclass may have several enclosing namespaces, one for each superclass it transitively extends. Prefix types provide an unambiguous way to name enclosing classes and packages of a class. Using prefix types, there is no 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.

As an example, in Figure 1, the name `Visitor` is syntactic sugar for the type `base[this.class].Visitor`. The dependent class `this.class` represents the run-time class of the object referred to by `this`. The prefix package `base[this.class]` is the enclosing package of `this.class` that is a subpackage of `base`. Thus, if `this` is an instance of a class in the package `pair`, `base[this.class]` represents the package `pair`.

To ensure the type of a formal parameter can be bounded by a non-dependent type, dependencies between the types of formal parameters of a method or constructor must be acyclic. Thus a declaration such as the following would *not* be allowed:

```
void m(b.class a, a.class b);
```

In addition, the type of a field cannot be dependent on a field path; this restriction ensures that a field's type can be bounded by a non-dependent type, preventing type checking from being undecidable.

The type p.`class` is well-formed only if p is final. An occurrence of p.`class` when p is not final will cause a compile-time error. However, to improve expressiveness and to ease porting of Java programs to J&, a non-final local variable x used as an actual argument to a method call may be *implicitly coerced* to the type x.`class` if x is not assigned by any of the call's actual arguments, including the receiver. This condition ensures that on entry to the method, all types dependent on the formal corresponding to 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 Intersection types

Nested intersection of classes and packages in J& is provided by *intersection types* [38, 12]. An intersection type S `&` T inherits all members of its base namespaces S and T.

J& supports only *shared* multiple inheritance: when a subclass (or subpackage) inherits from multiple base classes, perhaps via inheritance induced by extension of multiple containing namespaces, the new subclass may inherit the same superclass from more than one immediate superclass; however, instances of the subclass will not contain multiple subobjects for the common superclass. In the example of Figure 1(c), `pair_and_sum.Visitor` inherits from `base.Visitor` only once.

```
class A {
  class B { }
  void m() { }
}

class A1 extends A {
  class B { }
  class C { }
  void m() { }
  void p() { }
}

class A2 extends A {
  class B { }
  class C { }
  void m() { }
  void p() { }
}

abstract class D extends A1 & A2 { }
```

Figure 3. Multiple inheritance with name conflicts

3.3 Name conflicts

Since an intersection class type does not have a class body in the program text, its inherited members cannot be overridden by the intersection itself; however, subclasses of the intersection may override members.

When two namespaces declare members with the same name, a *name conflict* may occur in their intersection. 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. This solution is also used for nonvirtual "`super`" calls. If the superclass is an intersection type, the call may be ambiguous. The ambiguity is resolved by up-casting the special receiver `super` to the desired superclass.

Finally, two or more intersected classes may declare methods 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 `A1 & A2` is considered *abstract*. Because it cannot

```
class C { void n() { ... } }

class A1 {
  class B1 ext C { }
  class B2 ext C { }
  void m() {
    new A1[this.class].B1() & A1[this.class].B2();
  }
}

class A2 extends A1 {
  class B1 extends C { void n() { ... } }
  class B2 extends C { void n() { ... } }
  // now B1 & B2 conflict
}
```

Figure 4. Conflicts introduced by late binding

override the abstract method, the intersection is also abstract and cannot be instantiated. Subclasses of the intersection type (`D`, in the example), must override `m` to resolve the conflict, or else also be declared abstract.

3.4 Anonymous intersections

An instance of an intersection class type `A&B` may be instantiated by explicitly invoking constructors of both `A` and `B`:

```
new A() & B();
```

This intersection type is *anonymous*. As in Java, a class body may also be specified in the `new` expression, introducing a new anonymous subclass of `A&B`:

```
new A() & B() { ... };
```

This subclass is not identical to `A&B`.

`A` and `B` may have a name conflict which causes their intersection to be abstract. In this case, a class body must be provided in order to resolve the conflict and to allocate an anonymous intersection type.

Further binding may also introduce name conflicts. For example, in Figure 4, `A1.B1` and `A1.B2` do not conflict, but `A2.B1` and `A2.B2` do conflict. Since the anonymous intersection in `A1.m` may create an intersection of these two conflicting types, it should not be allowed. Because the type being instantiated is statically unknown, it is a compile-time error to instantiate an anonymous intersection of two or more dependent types; only anonymous intersections of non-dependent, non-conflicting classes are allowed.

3.5 Intersections of prefix types

Unlike with virtual classes [18], it is possible in J& 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 flexibility is needed for effective code reuse but complicates the definition of prefix types. Consider this example:

```
class A { class B { B m(); ... } }
class A1 extends A { class B { B x = m(); } }
class A2 extends A { class B { } }
class C extends A1.B & A2.B { }
```

As explained in Section 3.1, the unqualified name `B` in the body of class `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`. Since the method `m` and other code in `A.B` may be executed when `this` refers to an instance of `A1.B`, these two references to `B` should resolve to the same type; that is, it must be that `A[this.class]` is equivalent to `A1[this.class]`. This equivalence permits the assignment of the result of `m()` to `x` in `A1.B`. Similarly, the three types `A[C]`, `A1[C]`, and `A2[C]` should all be equivalent.

In general, if P' is a subtype of P, then P`[`T`]` and P'`[`T`]` should be equal. In J& the prefix type P`[`T`]` is defined as the intersection of all classes P' where $P'.C$ is a superclass of T, and P and P' share a common superclass. Thus, `A[C]` is the intersection of `A`, `A1`, and `A2`, that is, `A1&A2` $\approx$ `A[C]` $\approx$ `A1[C]` $\approx$ `A2[C]`.

Prefix types impose some restrictions on which types may be intersected. If two classes T_1 and T_2 contain conflicting methods, then their intersection is abstract, preventing the intersection from being instantiated. If T_1 or T_2 contain member classes, a prefix type of a dependent class bounded by one of these member classes could resolve to the intersection $T_1 \& T_2$. To prevent these prefix types from being instantiated, all member classes of an abstract intersection must also be abstract.

3.6 Constructors

As in Java, J& initializes objects using constructors. Since J& permits instances of dependent types to be allocated, the class being allocated may not be statically known. Constructors in J& are inherited and overridden like methods, allowing the programmer to invoke a constructor of a statically known superclass of the class being allocated.

The purpose of a constructor is to initialize the fields of an object. When a `final` field is added in a subclass, that field must be initialized. It is therefore required to either provide a field initializer or to override all inherited constructors to ensure the new field is initialized.

Constructor inheritance is limited to ensure fields can be initialized to meaningful values. For example, if a constructor were inherited by all subclasses, then, since all classes extend `Object`, all classes that add fields would have to override the default `Object()` constructor to initialize their new fields with a value passed into the constructor, which the `Object()` constructor cannot provide. Instead, constructors are inherited only via induced inheritance when the container of the constructor's class is extended; that is, the class $T'.C$ inherits constructors from $T.C$ when T is a superclass of T', but not from other superclasses of $T'.C$. Since a dependent class p`.class` may represent any subclass of p's statically known type, a consequence of this restriction is that p`.class` can only be explicitly instantiated if p's statically known class is `final`; in this case, since p`.class` is guaranteed to be equal to that `final` class, a constructor with the appropriate signature exists. The restriction does not prevent nested classes of dependent classes from being instantiated.

A constructor for a given class must invoke, using super constructor calls, a constructor of the class's declared superclass. If the superclass is an intersection type, it must invoke a constructor of each class in the intersection. Because of multiple inheritance, super constructors are invoked by explicitly naming them rather than using the `super` keyword as in Java. In Figure 5, `B.C` invokes the constructor of its superclass `A` by name.

Because J& implements shared multiple inheritance, an intersection class may inherit more than one subclass of a shared superclass. Invoking a shared superclass constructor more than once may lead to inconsistent initialization of `final` fields, possibly causing a run-time type error if the fields are used in dependent classes. There are two cases, depending on whether the intersection inherits one invocation or more than one invocation of a shared constructor.

In the first case, if all calls to the shared superclass's constructor originate from the same call site, which is multiply inherited into the intersection, then every call to the shared constructor will pass it the same arguments. In this case, the programmer need do

```
class A { A(int x); }
class B {
  class C extends A { C(int x) { A(x+1); } }
}
class B1 extends B {
  class C extends A { void m(); }
}
class B2 extends B { }
  class C extends A { void p(); }
}
class D extends B1 & B2 { }
```

Figure 5. Constructors of a shared superclass

nothing; the operational semantics of J& will ensure that the shared constructor is invoked exactly once.

For example, in Figure 5, 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(int)` 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. Similarly, if the programmer invokes:

```
new (B1 & B2).C(1);
```

there is only one call to the `A(int)` constructor and no conflict.

If, on the other hand, the intersection contains more than one call site that invokes a constructor of the shared superclass, or of the intersection itself is instantiated so that more than one constructor is invoked, then the programmer must resolve the conflict by specifying the arguments to pass to the shared superclass's constructor. The call sites inherited into the intersection will *not* be invoked. It is up to the programmer to ensure that the shared superclass is initialized in a way that is consistent with how its subclasses expect the object to be initialized. If the conflict is not resolved, the intersection cannot be instantiated.

In Figure 5, if one or both of `B1` and `B2` were to override the `C()` constructor, then since `B1.C` and `B2.C` have different constructors 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(int)` should invoke the constructor of `A`. This behavior is similar to that of constructors of shared virtual base classes in C++.

There would also be a conflict if the programmer were to invoke:

```
new B1.C(1) & B2.C(2);
```

The `A(int)` constructor would be invoked twice with different arguments. Thus, this invocation is illegal; however, since `B1.C&B2.C` is equivalent to `(B1&B2).C`, the intersection can be instantiated using that type, as shown above.

3.7 Type substitution

Because types may depend on final access paths, type checking of method calls requires substitution of the actual arguments for the formal parameters. A method may have a formal parameter whose type depends upon another parameter, including `this`. The actual arguments must reflect this dependency. For example, the class `base.Abs` in Figure 1 contains the following call:

```
v.visitAbs(this_A);
```

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

```
void visitAbs(base[this_V.class].Abs a);
```

```
package pair;

class TgtExp = base.Exp;
class Rewriter {
  TgtExp rewrite(Exp e)
    { ... }
}
```

```
package pair_and_sum
        extends pair;

class TgtExp = pair.Exp;
class Rewriter {
  TgtExp rewrite(Exp e)
    { ... }
}
```

Figure 6. Static virtual types

For clarity, each occurrence of `this` has been labeled with an abbreviation of its declared type. Since the formal type `base[this_V.class].Abs` depends on the receiver `this_V`, the type of the actual argument `this_A` must depend on the receiver *v*.

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[this_A.class].Visitor`. Substituting this type for `this_V.class` in the formal parameter type `base[this_V.class].Abs` yields `base[base[this_A.class].Visitor].Abs`, which is equivalent to `base[this_A.class].Abs`.

To ensure soundness, care must be taken when defining type substitution. The type substitution semantics described here generalize the original Jx substitution rules [32] to allow more expressivity. If the type of `v` were `base.Visitor`, then `v` might refer at run time to a `pair.Visitor` while at the same time `this_A` refers to a `base.Abs`. Substitution would yield `base[base.Visitor].Abs`, which simplifies to `base.Abs`. Since `base[this_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 `this_A` and `v` both have the same enclosing `base` package.

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. Substituting the type `base[this_A.class].Visitor`. for `this_V.class` is permitted since both `base[this_V.class]` and `base[this_A.class]` are exact. However, substituting `base.Visitor` for `this_V.class` is illegal since `base` is not exact; therefore, a call to `visitAbs` where `v` is declared to be a `base.Visitor` is not permitted.

Type substitution is performed when checking constructor calls and `new` expressions, as well. When checking allocation of a class `C`, the type `new C.class` is substituted in for `this` in the formal parameter types.

3.8 Static virtual types

Dependent classes and prefix types enable classes nested within a given containment hierarchy of packages to refer to each other without statically binding to a particular fixed package. This allows derived packages to further bind a class while preserving its relationship to other classes in the package. It is often useful to refer to other classes *outside* the class's containment hierarchy without statically binding to a particular fixed package. J& provides *static virtual types* to support this feature. Unlike virtual types in BETA [27], a static virtual type is an attribute of an enclosing package or class rather than of an enclosing object.

In Figure 6, the package `pair` declares a static virtual type `TargetExp` representing an expression of the target language of a rewriting pass, in this case an expression from the `base` compiler. The `rewrite` method takes an expression with type `pair[this.class].Exp` and returns a `base.Exp`. The `pair_and_sum` package extends the `pair` package and further

binds `TargetExp` to `pair.Exp`. A static virtual type can be further bound to any subtype of the original bound. Because `pair_and_sum.TargetExp` is bound to `pair.Exp`, the method `pair_and_sum.Rewriter.rewrite` must return a `pair.Exp`, rather than a `base.Exp` as in `pair.Rewriter.rewrite`.

With intersections, a static virtual type may be inherited from more than one superclass. Consider the following declarations:

```
class A { }
class A1 extends A { }
class A2 extends A { }

class B { class T = A; }
class B1 extends B { class T = A1; }
class B2 extends B { class T = A2; }
```

`B1&B2` inherits `T` from both `B1` and `B2`. The type `(B1&B2).T` must be a subtype of both `A1` and `A2`; thus, `(B1&B2).T` is bound to `A1&A2`.

To enforce the requirement that exactness be preserved by substitution, described in Section 3.7, static virtual types can be declared `exact`. For a given container namespace T, all members of the `exact` virtual type $T.C$ are of the same fixed run-time class or package. Exact virtual types can be further bound in a subtype of their container. For example, consider these declarations:

```
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` and not any subtype of `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`.

3.9 Packages

J& 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. In this section, we sketch the design of such a compiler. Most of the design described here was used in our port to J& of the Polyglot compiler framework [33], discussed in Section 6, but to maintain backward compatibility with the Java version of Polyglot, not all of the design was implemented.

The `base` package and packages nested within it contain all compiler code for the base language: Java, in the Polyglot framework. The nested packages `base.ast`, `base.types`, and `base.visit` contain classes for AST nodes, types, and visitors that implement compiler passes, respectively. All AST nodes are subclasses of `base.ast.Node`; most compiler passes are implemented as subclasses of `base.visit.Visitor`.

4.1 Orthogonal extension

Scalable, orthogonal extension of the base compiler with new data types and new operations is achieved through nested inheritance. To extend the compiler with new syntax, the `base` package is extended and new subclasses of `Node` can be added to the `ast` package.

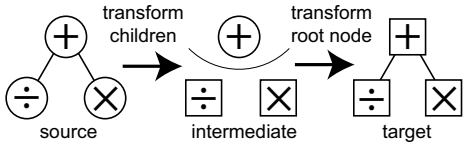

Figure 7. AST transformation

New passes can be added to the compiler by creating new `Visitor` subclasses.

Because the Visitor design pattern [20] is used to implement compiler passes, when a new AST node class is added to an extension's `ast` package, a `visit` method for the class must be added to the extension's `visit.Visitor` class. Because the classes implementing the compiler passes extend `base[this.class].visit.Visitor`, this `visit` method is inherited by all `Visitor` subclasses in the extension. Visitor classes in the framework can transform the AST by returning new AST nodes. The `Visitor` class implements default behavior for the `visit` method by simply returning the node passed to it, thus implementing an identity transformation. Visitors for passes affected by the new syntax can be overridden to support it.

4.2 Composition

Independent compiler extensions can be composed using nested intersection 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: the `main` method needs to be overridden in the composing extension to specify the order in which passes inherited from the composed extensions should run.

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.

4.3 Extensible rewriters

One challenge for building an extensible compiler is to implement transformations between different program representations. For example, 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. For a given transformation between representations in the base compiler, compiler extensions need to be able to scalably and modularly extend both a source and target representations and the transformation itself. However, it should not be necessary to change a transformation pass if the data extensions do not interact with the transformation.

For example, consider an abstract syntax tree (AST) node representing a binary operation. As illustrated in Figure 7, 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 the pass 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 or attributes.

```
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 8. Extensible rewriting example

In the `pair` compiler of Figure 1, the `TranslatePairs` pass transforms `pair` AST nodes into `base` AST nodes. If this compiler pass is reused in a compiler in which expressions have, say, 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.

Static virtual types, introduced in the Section 3.8, are used 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 explicitly represent nodes in the intermediate form as trees with a root in the source language but children in the target language, corresponding to the middle tree of Figure 7. This design is shown in Figure 8. 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` packages are then extended to create `ast` packages for the actual AST nodes. Finally, a package is created *inside each visitor class* for the intermediate form nodes of that visitor's specific source and target language.

Static virtual packages to link the packages together In the `ast_struct` package, children of each AST node reside in a `child` virtual package. The `ast` package extends the `ast_struct` package and further binds `child` to the `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 further binds `child` 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.

Both the `child` and `target` virtual packages are declared to be `exact`. This ensures that the children of a `tmp` node are in the `target` package itself (in this case `base.ast`) and not a derived package of the target (e.g., `pair.ast`).

5. Implementation

We implemented the J& compiler in Java using the Polyglot framework [33]. The compiler is a 2700-LOC (lines of code, excluding blank and comment lines) extension of the Jx compiler [32], itself a 22-kLOC extension of the Polyglot base Java compiler.[1]

[1] We have not yet implemented some language features such as package inheritance, anonymous intersection instantiation and `super` constructor calls for intersection classes. The LOC counts and the results in Section 6 were obtained with an earlier, less efficient, but more functionally complete, implementation.

J& is implemented as a translation to Java. The amount of code produced by the translation is proportional to the size of the source code. The translation does not duplicate code to implement inheritance. Class declarations are generated only for *explicit classes*, those classes (and interfaces) declared in the source program. For intersection types and *implicit classes*, classes inherited from another namespace but not further bound, data structures for method dispatching and run-time type discrimination for these classes are constructed on demand at run time.

5.1 Translating classes

Each explicit J& class is translated into four classes: an instance class, a subobject class, a class class, and a method interface. Figure 9 shows a simplified fragment of the translation of the code in Figure 1. Several optimizations, discussed below, are not shown.

At run time, each instance of a J& class T is represented as an instance of T's instance class, $\mathsf{IC}(T)$. Each explicit class has its own instance class. The instance class of an implicit class or intersection class is the instance class of one of its explicit superclasses. An instance of $\mathsf{IC}(T)$ contains a reference to an instance of the *class class* of T, $\mathsf{CC}(T)$. The class class contains method and constructor implementations, static fields, and type information needed to implement `instanceof`, prefix types, and type selection from dependent classes. If J& were implemented natively or had virtual machine support, rather than being translated to Java, then the reference to $\mathsf{CC}(T)$ could be implemented more efficiently as part of $\mathsf{IC}(T)$'s method dispatch table. All instance classes implement the interface `JetInst`.

Subobject classes and field accesses. Each instance of $\mathsf{IC}(T)$ contains a *subobject* for each explicit superclass of T, including T itself if it is explicit. The subobject class for an explicit class T contains all instance fields declared in T; it does not contain inherited fields. The instance class maintains a map from each explicit superclass of T to the subobject for that superclass. The static `view` method in the subobject class implements the map lookup function for that particular subobject. If J& were implemented natively, the subobjects could be inlined into the instance class and implemented more efficiently.

To get or set a field of an object, the `view` method is used to lookup the subobject for the superclass that declared the field. The field can then be accessed directly from the subobject. The `view` method could be inlined at each field access, but this would make the generated code more difficult to read and debug.

Class classes and method dispatch. For each J& class, there is a singleton class class object that is instantiated when the class is first used. A class class declaration is created for each explicit J& class. For an implicit or intersection class T, $\mathsf{CC}(T)$ is the runtime system class `JetClass`; the instance of `JetClass` contains a reference to the class class object of each immediate superclass of T.

```
package base;

// method interfaces for Exp
interface Exp$methods {
  interface Accept
    { JetInst accept(JetInst self, JetInst v); }
}

// class class of Exp
class Exp$class implements Exp$methods.Accept {
  JetInst accept(JetInst self, JetInst v)
    { /* cannot happen */ }
  static JetInst accept$disp(JetClass c, JetInst self,
                             JetInst v) {
    JetClass r = ... // find the class class with the
                     // most specific implementation
    return ((Exp$methods.Accept)r).accept(self, v);
  }
  ...
}

// class class of Abs
class Abs$class implements Exp$methods.Accept {
  JetInst accept(JetInst self, JetInst v) {
    Abs$ext.view(self).e =
      Exp$class.accept$disp(null, Abs$ext.view(self).e, v);
    return Visitor$class.visitAbs$disp(null, v, self);
  }
  ...
}

// instance class of Abs
class Abs implements JetInst {
  JetSubobjectMap extMap; // subobject map
  JetClass jetGetClass()
    { /* get the class class instance */ }
  ...
}

// subobject class of Abs
class Abs$ext {
  String x; JetInst e;
  static Abs$ext view(JetInst self) {
    // find the subobject for Abs in self.extMap
  }
}

...
```

Figure 9. Fragment of translation of code in Figure 1

The class class 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 object of prefix types and member types, including static virtual types. The code generated for expressions that dispatch on a dependent class (a `new x.class()` expression, for example) evaluates the dependent class's access path (i.e., `x`) and uses the method `jetGetClass()` to locate the class class object for the type.

All methods, including static methods, are translated to instance methods of the class class. This allows static methods to be invoked on dependent types, where the actual run-time class is statically unknown. Nonvirtual `super` calls are implemented by invoking the method in the appropriate class class instance.

Each method has an interface nested in the *method interface* of the J& class that first introduced the method. The class class implements the corresponding interfaces for all methods it declares or overrides. The class class of the J& class that introduces a method `m` also contains a method `m$disp`, responsible for method dispatching. The receiver and method arguments as well as a class class are passed into the dispatch method. The class class argument is used to implement nonvirtual `super` calls; for virtual calls, `null` is passed in (to prevent the receiver from being evaluated more than once) and the receiver's class class is used.

Single-method interfaces allow us to generate code only for those methods that appear in the corresponding J& class. An alternative, an interface containing all methods declared for each class, would require class classes to implement trampoline methods to dispatch methods they inherit but do not override, greatly increasing the size of the generated code.

As shown in Figure 9, all references to J& objects are of type `JetInst`. The translation mangles method names handle overloading. Name mangling is not shown in Figure 9 for readability.

Allocation. A factory method in the class class is generated for each constructor in the source class. The factory method for a J& class T first creates an instance of the appropriate instance class, and then initializes the subobject map for T's explicit superclasses, including T itself. Because constructors in J& can be inherited and overridden, constructors are dispatched similarly to methods.

Initialization code in constructors and initializers are factored out into initialization methods in the class class and are invoked by the factory method. A super constructor call is translated into a call to the appropriate initialization method of the superclass's class class.

5.2 Translating packages

To support package inheritance and composition, a package `p` is represented as a *package class*, analogous to the class class. The package class provides type information about the package at run time and access to the class class or package class instances of its member types. The package class of `p` is a member of package `p`. Since packages cannot be instantiated and contain no methods, package classes have no analogue to instance classes, subobject classes, or method interfaces.

5.3 Java compatibility

To leverage existing software and libraries, J& classes can inherit from Java classes. The compiler ensures that every J& class has exactly one most specific Java superclass. When the J& class is instantiated, there is only one `super` constructor call to some constructor of this Java superclass.

In the translated code, the instance class $\mathsf{IC}(T)$ is a subclass of the most specific Java superclass of T. When assigning into a variable or parameter that expects a Java class or interface, the instance of $\mathsf{IC}(T)$ can be used directly. A cast may need to be inserted because references to $\mathsf{IC}(T)$ are of type `JetInst`, which may not be a subtype of the expected Java type; these inserted casts always succeed. The instance class also overrides methods inherited from Java superclasses to dispatch through the appropriate class class dispatch method.

5.4 Optimizations

One problem with the translation described above is that a single J& object is represented by multiple objects at run time: an instance class object and several subobjects. This slows down allocation and garbage collection.

A simple optimization is not to create subobjects for those J& classes that do not introduce instance fields. The instance class of explicit J& class T can inline the subobjects into $\mathsf{IC}(T)$. Thus, at run time, an instance of an explicit J& class can be represented by a single object; an instance of an implicit class or intersection class is represented by an instance class object and subobjects for superclasses not merged into the instance class object. We expect this optimization to improve efficiency greatly.

Name	Extends Java 1.4 ...	LOC original	LOC ported	% original
`polyglot`	with nothing	31888	27984	87.8
`param`	with infrastructure for parameterized types	513	540	105.3
`coffer`	with resource management facilities similar to Vault [13]	2965	2642	89.1
`j0`	with pedagogical features	679	436	64.2
`pao`	to treat primitives as objects	415	347	83.6
`carray`	with constant arrays	217	122	56.2
`covarRet`	to allow covariant method return types	228	214	93.9

Table 1. Ported Polyglot extensions

	`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

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 J&. 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 [32]; more than a dozen research projects have used Polyglot to implement various extensions to Java (e.g., JPred [31], JMatch [26], as well as Jx and J&). For this work we ported six extensions ranging in size from 200 to 3000 LOC.

The extensions are summarized in Table 1. The parsers for the base compiler, extensions, and compositions were generated from CUP [22] 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.

The port of the base compiler was our first attempt to port a large program to J&, and was completed by one of the authors within a few days, excluding time to fix bugs in the J& compiler. 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 which were needed to make the Java version extensible, but which are not needed in J&. We eliminated only extension patterns that were obviously unnecessary, and could remove additional code with more effort.

The number of type downcasts in each compiler extension is reduced in J&. 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 pattern 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`.

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.

6.2 Pastry

We also ported the FreePastry peer-to-peer framework [39] version 1.2 to J& 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 LOC.

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 J& 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. Beehive and PC-Pastry extend Pastry with caching functionality [36]. PC-Pastry 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 nested intersection, 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 con-

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

figuration 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 J& 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 dispatching 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 attributable to moving code from the CorONA extension to the CorONA–Beehive composition.

6.3 Porting Java to J&

Porting Java code to J& was usually straightforward, but certain common issues are worth discussing.

Type names. In J&, 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 J&, 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 so they could 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, the type system ensures its new value is a class enclosed by the same runtime 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 (e.g., `this.thePastryNode.class`) than the Polyglot port. Several casts needed to be inserted in the J& 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. 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 J& are similar to virtual classes [27, 28, 23, 18]. Virtual classes were originally developed for the language BETA [27, 28], primarily for generic programming rather than for extensibility.

Although virtual classes in BETA were not statically type safe, Ernst's generalized BETA (gbeta) language [14, 15] uses path-dependent types, similar to dependent classes in J&, to ensure static type safety. Type-safe virtual classes using path-dependent types were formalized by Ernst et al. in the *vc* calculus [18].

A key difference between J&'s nested classes and virtual classes is that virtual classes are attributes of an object, called the enclosing instance, rather than attributes of a class. Virtual classes may only have one enclosing instance. For this reason, a virtual class can extend only other classes nested within the same object; it may not extend a more deeply nested virtual class. This can limit the ability to extend components of a larger system. Because it is unique, the enclosing instance of a virtual class can be referred to unambiguously with an `out` path: `this.out` is the enclosing instance of `this`'s class. In contrast, J& uses prefix types to refer to enclosing classes.

Both J& and gbeta provide virtual superclasses, the ability to late-bind a supertype declaration. When the containing namespace of a set of classes is extended via inheritance, the derived namespace replicates the class hierarchy of the original namespace, forming a *higher-order hierarchy* [17]. Unlike in J&, because virtual classes are contained in an object rather than in a class, there is no subtyping relationship between classes in the original hierarchy and further bound classes in the derived hierarchy. There is an induced subclass relationship, however.

The gbeta language supports multiple inheritance. As in J&, commonly named virtual classes inherited into a class are themselves composed [15]. However, multiple inheritance is limited to other classes nested within the same enclosing instance.

In gbeta, each object defines a *family* of classes: the collection of mutually dependent virtual classes nested within it. Virtual classes in gbeta support *family polymorphism* [16]: two virtual classes enclosed by distinct objects cannot be statically confused. When a containing namespace is extended, family polymorphism ensures the static type safety of the classes in the derived family by preventing it from treating classes belonging to the base family as if they belonged to the extension. Because nested classes in J& are attributes of their enclosing class, rather than an enclosing object, J& supports nested inheritance supports what Clarke et al. [10] call *class-based family polymorphism*. With virtual classes, all members of the family are named from a single "family object", which must be accessible throughout the system. In contrast, 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.

Tribe [10] is another language that provides a variant of virtual classes. By treating a final access path p as a type, nested classes in Tribe can be considered attributes of an enclosing class as in Jx and J& or as attributes of an enclosing instance as in BETA and its derivatives. This flexibility allows a further bound class to be a subtype of the class it overrides, like in J& but unlike with virtual classes. Tribe also supports multiple inheritance. However, superclasses of a Tribe class must be nested within the same enclosing class, limiting extensibility. This restriction allows the enclosing type to be named using an `owner` attribute: T`.owner` is the enclosing class of T.

Concord [24] 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 J&. Relative supertype declarations provide functionality similar to virtual superclasses. Groups in Concord cannot be nested, nor can groups be multiply inherited.

Multiple inheritance. J& provides multiple inheritance through nested intersection. Intersection types were introduced by Reynolds in the language Forsythe [38] and were used by Compagnoni and Pierce to model multiple inheritance [12]. Cardelli [8] presents a formal semantics of multiple inheritance.

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++ [42] and Self [9], treat all name conflicts as ambiguities to be resolved by the caller. Some languages [29, 3, 40] allow methods to be renamed or aliased.

A *mixin* [3, 19], 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 of classes. Mixins can be simulated using explicit multiple inheritance. J& also provides additional mixin-like functionality through virtual superclasses.

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.

Traits [40] are collections of abstract and non-abstract methods that may be composed with state to form classes. Since traits do not have fields, many of the issues introduced by multiple inheritance (for example, whether to duplicate code inherited through more than one base trait) are avoided. The code reuse provided by traits is largely orthogonal to that provided by nested inheritance and could be integrated into J&.

Scala 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 J&, 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 J& more flexibility, while preserving type soundness. Scala provides virtual types, but not virtual classes. It has no analogue to prefix types, nor does it provide virtual superclasses, limiting the scalability of its extension mechanisms. Scala supports composition using traits. Since traits do not have fields, new state cannot be easily added into an existing class hierarchy.

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. Bruce and Vanderwaart [7, 5] propose *type groups* as a means to aggregate and extend mutually dependent classes, similarly to Concord's group construct, but using matching rather than subtyping.

Open classes. An *open class* [11] 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 are late bound to refined versions of those types and methods. This feature provides similar functionality to the late binding of types provided by `this`-dependent classes and prefix types in J&.

Since reuse is based on import of classboxes rather than inheritance, classboxes do not support multiple inheritance, but they do allow multiple imports. When two classboxes that both refine the same class are imported, the classes are not composed like in J&. Instead, one of the classes is chosen over the other.

Class hierarchy composition. Tarr et al. [43] define a specification language for composing class hierarchies. Rules specify how to merge "concepts" in the hierarchies. Nested intersection supports composition with a rule analogous to merging concepts by name.

Snelting and Tip [41] 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 of classes in the hierarchies. J& reports a conflict if composed class hierarchies have a *static interference*, but makes no effort to detect dynamic interference.

Aspect-oriented programming. Aspect-oriented programming (AOP) [25] 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 [30] is an aspect-oriented language that also supports family polymorphism, permitting application of aspects to mutually recursive nested types.

8. Conclusions

This paper introduces nested intersection 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 static virtual types offer important expressive power.

Nested intersection has been implemented in an extension of Java called J&. Using J&, 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 com-

pilers by choosing from available language implementation components. We demonstrated the utility of nested intersection outside the compiler domain by porting the FreePastry peer-to-peer system to J&. The effort required to port Java programs to J& 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 J&. A formal treatment with a sketch of a proof of soundness can be found in Appendix A.

Nested intersection 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, Ruijie Wang, and Lantian Zheng provided useful feedback on various drafts of this paper. Thank you to Michael Clarkson for his very detailed comments and for the pun. 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] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in Self. *Lisp and Symbolic Computation*, 4(3):207–222, June 1991.

[10] Dave Clarke, Sophia Drossopoulou, James Noble, and Tobias Wrigstad. Tribe: More types for virtual classes. Submitted for publication. Available at `http://slurp.doc.ic.ac.uk/pubs.html`, December 2005.

[11] 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.

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

[22] 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/`.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] O. Lehrmann Madsen, B. Møller-Pedersen, and K. Nygaard. *Object Oriented Programming in the BETA Programming Language*.

Addison-Wesley, June 1993.

[28] 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.

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

[30] 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.

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

[32] 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.

[33] Nathaniel Nystrom, Michael R. 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 Gün 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 [21], 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] 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.

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

[43] 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.

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

[45] 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	$p ::= v \mid x \mid p.f \mid \texttt{new}\ T$
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	$\Gamma ::= \emptyset \mid \Gamma, x : T$

Figure 10. Grammar

A. Formal semantics

This section presents a formal semantics for the core J& type system and sketches a soundness proof for the semantics. To reduce complexity and to save space, several features including package inheritance, constructors, and static virtual types are not modeled in the semantics.

A grammar for the calculus is shown in Figure 10. We use the notation $\overline{a}$ for the set $a_1, \ldots, a_n$ for $n \geq 0$. The length of $\overline{a}$ is written $\#(\overline{a})$. A term with a list subterm should be interpreted as a list of terms; for example, $\overline{f} = \overline{e}$ should be read $f_1 = e_1, \ldots, f_n = e_n$.

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 contains 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 a method body; formal parameters are final.

Types T are either names of top-level classes C, nested classes $T.C$, dependent classes $p.\texttt{class}$, prefix types $P\texttt{[}T\texttt{]}$, or intersection types $\texttt{\&}\overline{T}$. The intersection type $\texttt{\&}\overline{T}$ can be read $T_1\ \texttt{\&}\ \ldots\ \texttt{\&}\ T_n$. Non-dependent types are written S and class names are written P. In the calculus, the prefix type $P\texttt{[}T\texttt{]}$ 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 desugaring 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, a final field access $p.f$, or $\texttt{new}\ T$. Expressions are values, parameters x, field accesses, field assignments, 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 Class lookup

The class table, CT, defined in Figure 11, maps class names P to class declarations. We write $CT(P) = \bot$ if P has no definition. The judgment $\vdash S\ \mathsf{defined}$ states that S is a well-formed non-dependent type; the judgment holds when either S is a class in the class table, further binds a defined class, or all its members are defined. 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 $\mathsf{mem}(S)$.

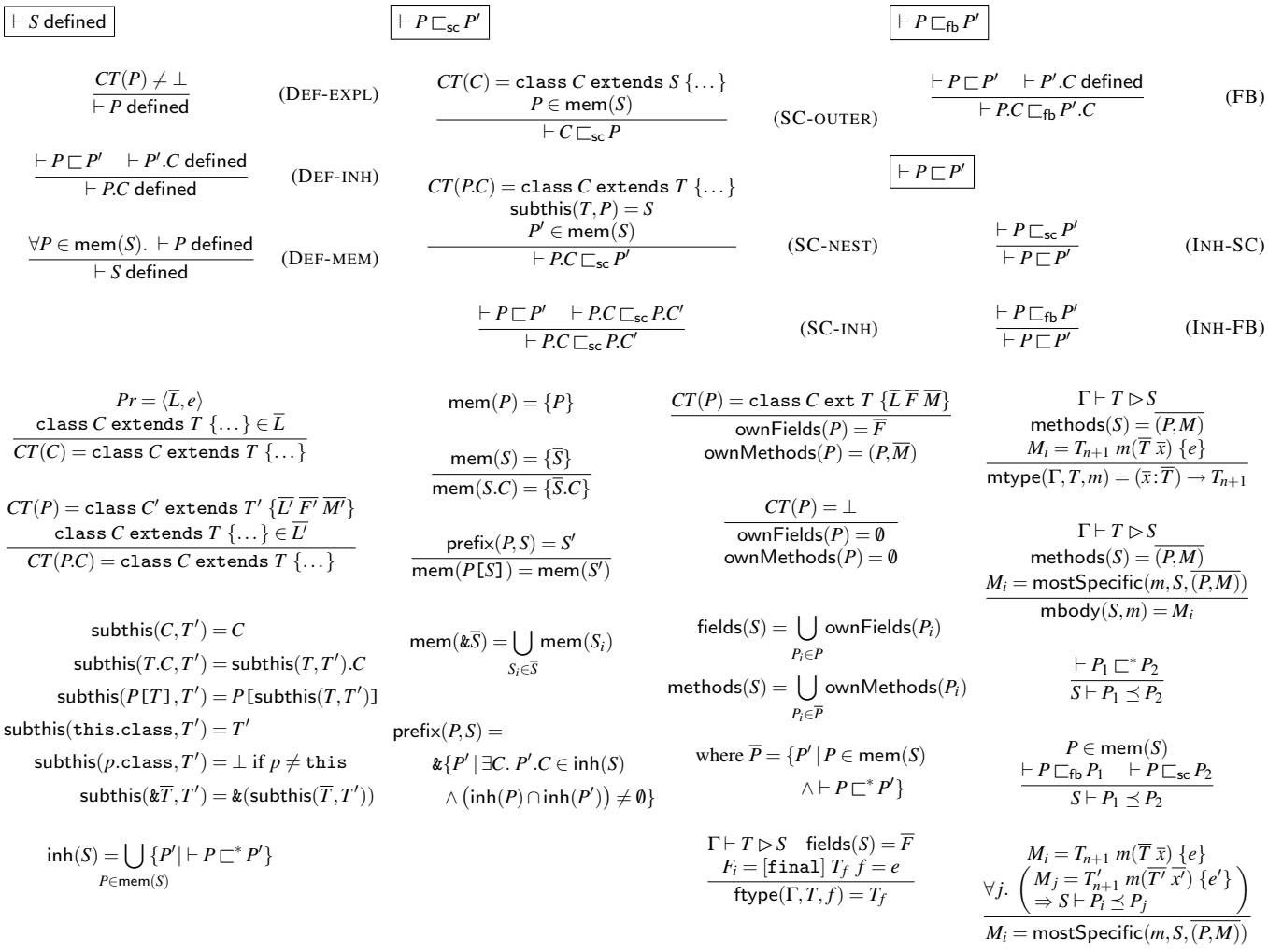

Figure 11. Subclassing and auxiliary functions

A.2 Subclassing and further binding

Inheritance among classes is defined in Figure 11. The rules are similar to those defined for the language Tribe [10]. 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`. No other access paths are allowed in superclass declarations. SC-INH defines subclassing for implicit classes not in the domain of the class table.

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. We write $\vdash P \sqsubset P'$ if P either subclasses or further binds P'; $\sqsubset^*$ is the reflexive transitive closure of $\sqsubset$. The function $\mathsf{inh}(S)$ returns the set of all superclasses of S, including S itself.

A.3 Prefix types

The meaning of non-dependent prefix types $P[S]$ is defined by the prefix function in Figure 11. 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. This definition ensures that if P is a subtype of P', then $P[S]$ is equal to $P'[S]$, as desired in Section 3.5.

A.4 Member lookup

Method and field lookup functions are shown in Figure 11. 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, 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 the $\mathsf{fields}(S)$ 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 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. As in Jx, further bound classes $P.C$ are ordered before declared superclasses.

A.5 Type well-formedness and simple bounds

The judgment $\Gamma \vdash T \rhd S$ in Figure 12 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.

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

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

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

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

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

$$\frac{\Gamma \vdash T\{\!\{T_s/x\}\!\} = T'}{\Gamma \vdash \mathtt{new}\ T.\mathtt{class}\{\!\{T_s/x\}\!\} = \mathtt{new}\ T'.\mathtt{class}}$$

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

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

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

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

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

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

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

Figure 13. Type substitution

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.6 Typing

The judgment $\Gamma \vdash_{\mathsf{fin}} p : T$ in Figure 12 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 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 13. A type T is exact, written $\mathsf{exact}(T)$, if it is a dependent class a prefix of an exact type, or an intersection containing an exact type. Type substitution for field accesses need not preserve exactness of the field type. In contrast, the rule T-SET does require that exactness be preserved when substituting 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. Substitution of the return type need not preserve exactness.

A `new` expression is well-typed if all it initializes only declared fields of a well-formed type. 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 $p.\mathtt{class}$, and T-SUB is the standard subsumption rule.

A.7 Subtyping and type equivalence

Subtyping rules are defined in Figure 12. 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. Following the semantics of Tribe [10], 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 by S-NULL.

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

S-MEET-LB and S-MEET-G are from Compagnoni and Pierce [12] and define subtyping for intersection types. Together these two rules imply that intersection types are associative and commutative and that the singleton intersection type $\&T$ is equivalent to its element type T. With the other rules above, these rules also imply the intuitive judgments $\Gamma \vdash P[\&\overline{T}] \leq P[T_i]$ and $\Gamma \vdash (\&\overline{T}).C \leq T_i.C$.

Rule S-NEW-BD is needed to preserve the type of a $\mathtt{new}\ T(\overline{f} = \overline{e})$ expression as T is evaluated. The rules S-NEW-NEW, S-NEW-LOC, and S-NEW-PRE define equivalence for types dependent on locations or $\mathtt{new}\ T$. All location-dependent classes are equal to `new`-dependent paths of the same type; thus, all location-dependent classes of the same type are equal to each other. Finally, a `new`-dependent class is equivalent to the appropriate prefix type of a `new`-dependent class.

A.8 Program typing

Program typing rules are presented in Figure 15. The P-OK says the program Pr is well-formed if all class declarations are well-formed, if the "`main`" expression is well-typed, and if the transitive closure of the inheritance relation $\sqsubset$ is acyclic.

By L-OK, 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` bound to the class's container (if any). Additionally, the class's container and superclass must be in the domain of subthis. This ensures that the only access path embedded in the superclass declaration is `this`. The class must also conform to all of its superclasses.

A class P conforms to P' if all of the following hold:

- If both P and P' have a member class D, then $P.D$'s declared superclass is a subtype of $P'.D$'s.
- The field names of P and P' are disjoint. This requirement simplifies the semantics by ensuring field names are unique. (We treat $\overline{F}$ as a map from field names to field declarations and write $\mathrm{dom}(\overline{F})$ for the set of field names.)

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

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

$\boxed{\Gamma \vdash T}$

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

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

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

$$\frac{\Gamma \vdash T}{\Gamma \vdash \mathtt{null} : T}\ (\text{T-NULL}) \qquad \frac{\Gamma \vdash S}{\Gamma \vdash \ell_S : S}\ (\text{T-LOC}) \qquad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}\ (\text{T-VAR}) \qquad \frac{\Gamma \vdash e : T_0 \quad \mathsf{ftype}(\Gamma, T_0, f) = [\mathtt{final}]\ T_f \quad \Gamma \vdash T_f\{\!\{T/\mathtt{this}\}\!\} = T'_f}{\Gamma \vdash e_0.f : T'_f}\ (\text{T-GET}) \qquad \frac{\Gamma \vdash e_0 : T_0 \quad \Gamma \vdash e_1 : T'_f \quad \mathsf{ftype}(\Gamma, T_0, f) = T_f \quad \Gamma \vdash T_f\{\!\{T_0/\mathtt{this}\}\!\}_{\times} = T'_f}{\Gamma \vdash e_0.f = e_1 : T'_f}\ (\text{T-SET})$$

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

$$\frac{\Gamma \vdash T \quad \Gamma \vdash \overline{e} : \overline{T} \quad \forall i.\ \mathsf{ftype}(\Gamma, T, f_i) = [\mathtt{final}]\ T_i}{\Gamma \vdash \mathtt{new}\ T(\overline{f} = \overline{e}) : \mathtt{new}\ T.\mathtt{class}}\ (\text{T-NEW}) \qquad \frac{\Gamma \vdash_{\mathsf{fin}} p : T}{\Gamma \vdash p : p.\mathtt{class}}\ (\text{T-FIN}) \qquad \frac{\Gamma \vdash e_1 \vdash T_1 \quad \Gamma \vdash e_2 \vdash T_2}{\Gamma \vdash e_1;\ e_2 : T_2}\ (\text{T-SEQ}) \qquad \frac{\Gamma \vdash e : T \quad \Gamma \vdash T \leq T'}{\Gamma \vdash e : T'}\ (\text{T-SUB})$$

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

$$\Gamma \vdash T \leq T\ (\text{S-REFL}) \qquad \frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash T_2 \leq T_3}{\Gamma \vdash T_1 \leq T_3}\ (\text{S-TRANS}) \qquad \frac{\Gamma \vdash T \leq P \quad CT(P) = \mathtt{class}\ C\ \mathtt{extends}\ T'\ \{\ldots\} \quad P = P'.C \Rightarrow \mathsf{subthis}(T', P'[T]) = T'' \quad P = C \Rightarrow T' = T''}{\Gamma \vdash T \leq T''}\ (\text{S-SUP}) \qquad \frac{\Gamma \vdash T \leq T' \quad \Gamma \vdash T'.C}{\Gamma \vdash T.C \leq T'.C}\ (\text{S-NEST})$$

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

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

$$\frac{\mathsf{mem}(S_1) = \mathsf{mem}(S_2)}{\Gamma \vdash \mathtt{new}\ S_1.\mathtt{class} \approx \mathtt{new}\ S_2.\mathtt{class}}\ (\text{S-NEW-NEW}) \qquad \Gamma \vdash \mathtt{new}\ S.\mathtt{class} \approx \ell_S.\mathtt{class}\ (\text{S-NEW-LOC}) \qquad \frac{\Gamma \vdash T \approx \mathtt{new}\ S'.\mathtt{class} \quad S = \mathsf{prefix}(P, S')}{\Gamma \vdash \mathtt{new}\ S.\mathtt{class} \approx P[T]}\ (\text{S-NEW-PRE})$$

Figure 12. Static semantics

- If both P and P' define a method m, then the method in P correctly overrides the method P'.

Method M in P correctly overrides M' if the number of formal parameters are equal, the parameter types of M are supertypes of the parameter types of M', and the return type of M is a subtype of M'. Subtyping checks are done with fresh names substituted in for the parameter names occurring in the types. Using the judgment $\vdash \Gamma\ \mathsf{ok}$, it is required that there be no cyclic dependencies between formal parameters; that is there is a permutation of the parameters such that the type of a parameter $i+1$ is well-formed when just the first i parameters are in scope.

Finally, field and method declarations are well-formed by rules F-OK and M-OK, respectively if the types occurring in the signatures well-formed and if the initializer is method body is well-typed.

A.9 Operational semantics

A small-step operational semantics is shown in Figure 16. 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 or $\mathsf{NullError}$ The initial configuration for program $\langle \overline{L}, e \rangle$ is $e, \emptyset$. Final configurations are of the form v, H, or $\mathsf{NullError}, H$.

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

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

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

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

Figure 14. Well-formed environments

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

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	$::= \&(\overline{S}, TE, \overline{T'})$
		$\mid TE.C$
		$\mid E.\texttt{class}$
		$\mid \texttt{new}\ TE.\texttt{class}$
		$\mid P[TE]$
null error contexts	NE	$::= \texttt{null}.f$
		$\mid \texttt{null}.f = e$
		$\mid \texttt{null}.m(\overline{e})$
		$\mid \texttt{new}\ TE[\texttt{null}](\overline{f} = \overline{e})$

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

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

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

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

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

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

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

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

Figure 16. Operational semantics

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 and set a field in a heap object, respectively. R-CALL uses the mbody function defined in Figure 11 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.10 Well-formed heaps

Figure 17 shows the heap typing rules. The judgment $H \vdash \ell_S$ states that a location ℓ_S is well-formed for a heap H if it maps to an object containing all declared fields of S and each value stored in those fields has the correct type and, if a location, is also well-formed in H. Rule H-NULL states that the `null` value is always well-formed.

A heap H is well-formed, written $\vdash H$, if all locations in its domain are well-formed. Finally, a configuration is well-formed, written $\vdash e, H$ if H is well-formed and all free locations of e, $\mathsf{locs}(e)$, are in H.

A.11 Soundness

To prove soundness we use the standard technique of proving subject reduction and progress lemmas [45]. The key lemmas are stated here.

LEMMA A.1. (Congruence) *If* $\Gamma \vdash E[e] : T$, $\Gamma \vdash e : T'$, *and* $\Gamma \vdash e' : T'$, *then* $\Gamma \vdash E[e] : T'$.

PROOF: By induction on the structure of E. □

LEMMA A.2. (Substitution) *If* $\Gamma, x : T, \Gamma' \vdash e : T'$ *and* $\emptyset \vdash v : T$, *then* $\Gamma, \Gamma'\{v/x\} \vdash e\{v/x\} : T'\{v/x\}$.

PROOF: By induction on the structure of e. □

LEMMA A.3. (Type substitution) *If* $\Gamma \vdash v : T_v$, *and* $\Gamma \vdash T\{\!\{T_v/x\}\!\} = T'$, *then* $\Gamma \vdash T\{v/x\} \leq T'$.

PROOF: By induction on structure of T. □

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

LEMMA A.4. (Subject reduction) *If* $\emptyset \vdash e : T$, $\vdash e, H$, *and* $e, H \longrightarrow r, H'$, *then either* $r = e'$ *and* $\emptyset \vdash e' : T$ *and* $\vdash e', H'$, *or* $r = \mathsf{NullError}$.

PROOF: By induction on the structure of e. e must either be a redex or equal to $E[e_0]$ for some E and e_0.

- Case $e = E[e_0]$: Then $r = E[e_0']$ and by R-CONG, $e_0, H \longrightarrow e_0', H'$. By the induction hypothesis, if $\emptyset \vdash e_0 : T_0$, then $\emptyset \vdash e_0' : T_0$ and $\vdash e_0', H'$. By Lemma A.1, since $\emptyset \vdash E[e_0] : T$, we have $\emptyset \vdash E[e_0'] : T$. Since $\vdash e_0', H'$, $\vdash H'$. Hence, $\vdash E[e_0'], H'$,
- Case $e = E[NE]$: Then by R-NULL, $r = \mathsf{NullError}$. Done.
- Case $e = \ell_S.f_i$: By R-GET, $r = v_i = H(\ell_S)[f_i]$. Let $\mathsf{ftype}(\emptyset, S, f_i) = T_i$. By T-GET, $\emptyset \vdash \ell_S.f_i : T_i\{\ell_S/\texttt{this}\}$. Since H is well-formed, by H-LOC $\emptyset \vdash H(\ell_S)[f_i] : T_i\{\ell_S/\texttt{this}\}$. Done.
- Case $e = \ell_S.f = v$: By R-SET, $e' = v$ and $\emptyset \vdash v : T$ follows trivially from T-SET.

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

$$\frac{\begin{array}{c} P.C \vdash \overline{L}\ \mathsf{ok} \quad P.C \vdash \overline{F}\ \mathsf{ok} \quad P.C \vdash \overline{M}\ \mathsf{ok} \\ P \vdash T\ \mathsf{super\ ok} \\ \forall P_i \in \mathsf{inh}(P.C).\ \vdash P.C \text{ conforms to } P_i \end{array}}{P \vdash \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}\ \mathsf{ok}} \tag{L-OK}$$

$$\frac{\begin{array}{c} (P \neq \mathsf{nil} \Rightarrow \mathtt{this} : P \vdash T\ \mathsf{ok}) \quad (P = \mathsf{nil} \Rightarrow \emptyset \vdash T\ \mathsf{ok}) \\ (P \neq \mathsf{nil} \Rightarrow \mathsf{subthis}(T, P) \neq \bot) \end{array}}{P \vdash T\ \mathsf{super\ ok}}$$

$$\frac{\begin{array}{c} CT(P) = \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\} \\ CT(P') = \mathtt{class}\ C'\ \mathtt{extends}\ T'\ \{\overline{L'}\ \overline{F'}\ \overline{M'}\} \\ \forall i, j.\ \begin{pmatrix} L_i = \mathtt{class}\ D\ \mathtt{extends}\ T_i\ \{\ldots\} \wedge \\ L'_j = \mathtt{class}\ D\ \mathtt{extends}\ T'_j\ \{\ldots\} \end{pmatrix} \Rightarrow \mathtt{this} : P \vdash T_i \leq T'_j \\ (\mathrm{dom}(\overline{F}) \cap \mathrm{dom}(\overline{F'})) = \emptyset \\ \forall i, j.\ \begin{pmatrix} M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \wedge \\ M_j = T'_{n+1}\ m(\overline{T'}\ \overline{x'})\ \{e'\} \end{pmatrix} \Rightarrow P \vdash M_i \text{ overrides } M'_j \end{array}}{\vdash P \text{ conforms to } P'}$$

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

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

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

Figure 15. Program typing

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

$$H \vdash \mathtt{null} \tag{H-NULL}$$

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

$$\frac{\vdash H \quad \mathsf{locs}(e) \subseteq \mathrm{dom}(H)}{\vdash e, H} \tag{CONFIG}$$

Figure 17. Well-formed heaps

Let $H(\ell_S) = S\ \{\overline{f} = \overline{v}$. Since $\vdash e, H$, we have $\vdash H$ and $H \vdash \overline{v}$ and also $H \vdash v$.

Let $\mathsf{ftype}(\emptyset, S, f) = T_f$. Then $T = T_f\{\ell_S/\mathtt{this}\}$. Since H' changes only $H(\ell_S)[f]$, but does not change its type and since $H \vdash v$, H' is also well-formed by H-LOC.

- Case $e = \ell_S.m(\overline{v})$: By T-CALL,
  - $\emptyset \vdash \ell_S : \ell_S.\mathtt{class}$
  - $\mathsf{mtype}(\emptyset, \ell_S.\mathtt{class}, m) = (\overline{x} : \overline{T}) \to T_{n+1}$
  - $\emptyset \vdash \overline{v} : \overline{T'}$
  - $\forall i.\ \emptyset \vdash T_i\{\!\{\mathtt{this}, \overline{x}/\ell_S.\mathtt{class}, \overline{T'}\}\!\}_{\mathsf{x}} = T'_i$
  - $\emptyset \vdash T_{n+1}\{\!\{\mathtt{this}, \overline{x}/\ell_S.\mathtt{class}, \overline{T'}\}\!\} = T$

  By R-CALL, $\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e'\}$. By M-OK, $\Gamma \vdash e' : T_{n+1}$ where $\Gamma = \mathtt{this} : P, \overline{x} : \overline{T}$ for some $\emptyset \vdash S \leq P$. By Lemma A.2, $\emptyset \vdash e'\{\ell_S/\mathtt{this}, \overline{v}/\overline{x}\} : T_{n+1}\{\ell_S/\mathtt{this}, \overline{v}/\overline{x}\}$, and by Lemma A.3, $\emptyset \vdash T_{n+1}\{\ell_S/\mathtt{this}, \overline{v}/\overline{x}\} \leq T$.
- Case $e = \mathtt{new}\ T(\overline{f} = \overline{v})$: By T-NEW, $\Gamma \vdash e : \mathtt{new}\ T.\mathtt{class}$. Let $e' = \mathtt{new}\ S(\overline{f} = \overline{v}, \overline{f'} = \overline{e'})$. By F-OK, $\Gamma \vdash e'_i : T'_i$; thus, by T-NEW $\Gamma \vdash e' : \mathtt{new}\ S.\mathtt{class}$. By S-NEW-BD $\Gamma \vdash \mathtt{new}\ S.\mathtt{class} \leq \mathtt{new}\ T.\mathtt{class}$. Thus, by T-SUB, $\Gamma \vdash e' : \mathtt{new}\ T.\mathtt{class}$.
- Case $e = \mathtt{new}\ S(\overline{f} = \overline{v})$: If $\#(\overline{f}) \neq \#(\mathsf{fields}(S))$, the previous case applies. Otherwise, $e' = \ell_S$. $\emptyset \vdash e' : S$ by T-LOC. Since $\vdash e, H$, we have $\vdash H$ and $H \vdash \overline{v}$. Therefore, H' is well-formed.
- Case $e = v$; e': Trivial by R-SEQ and T-SEQ.

□

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

PROOF: By cases on the structure of e. □

Soundness follows directly from the subject reduction and progress lemmas.

THEOREM A.6. (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 $e, \emptyset \longrightarrow^{*} r, H'$.*

PROOF: Follows from Lemma A.4 and Lemma A.5. □