Scalable Extensibility via Nested Inheritance

(Technical Report)

Nathaniel Nystrom Stephen Chong Andrew C. Myers

Computer Science Department

Cornell University

{nystrom,schong,andru}@cs.cornell.edu

Abstract

Inheritance is a useful mechanism for factoring and reusing code. However, it has limitations for building extensible systems. We describe *nested inheritance*, a mechanism that addresses some of the limitations of ordinary inheritance and other code reuse mechanisms. Using our experience with an extensible compiler framework, we show how nested inheritance can be used to construct highly extensible software frameworks. The essential aspects of nested inheritance are formalized in a simple object-oriented language with an operational semantics and type system. The type system of this language is sound, so no run-time type checking is required to implement it and no run-time type errors can occur. We describe our implementation of nested inheritance as an unobtrusive extension of the Java language, called Jx. Our prototype implementation translates Jx code to ordinary Java code, without duplicating inherited code.

1 Introduction

Conventional language mechanisms do not adequately support the reuse and extension of existing code. Libraries and module systems are perhaps the most widely used mechanisms for code reuse; a given library can be used by any code that respects its interface. Inheritance adds more power: it enables *frameworks*, class libraries that can be reused with some modifications or extensions. But these mechanisms do not adequately support our goal of *scalable extensibility*: the ability to extend a body of code while writing new code proportional to the differences in functionality.

In our work on the Polyglot extensible compiler framework [25], we found that ordinary object-oriented inheritance and method dispatch do not adequately support extensibility. Because inheritance operates on one class at a time, some kinds of code reuse are difficult or impossible. For example, inheritance does not support extension of an existing class library by adding a given field or method to all subclasses of a given class. Inheritance is also inadequate for extending a set of classes whose objects interact according to some protocol, a pattern that occurs in many domains ranging from compilers to user interface toolkits. It can be difficult to use inheritance to reuse and extend interdependent classes.

Nested inheritance is a language mechanism designed to support scalable extensibility. Nested inheritance creates an interaction between containment and inheritance. When a container (a namespace such as a class or package) is inherited, all of its components—even nested containers—are inherited too. In addition, inheritance and subtyping relationships among these components are preserved in the derived container. By deriving one container from another, inheritance relationships may be concisely constructed among many contained classes.

To avoid surprises when extending a base system, it is important that inherited code remain type-safe in its new context; further, type safety should be enforced statically. Nested inheritance supports sound compile-time type checking. This soundness is not easily obtained, because for extensibility, types mentioned in inherited code need to be interpreted differently in the new, inheriting context. Two new type constructs make sound reinterpretation of types possible: *dependent classes* and *prefix types*.

We have designed a new language, Jx, which adds nested inheritance to Java. Jx demonstrates that nested inheritance integrates smoothly into an existing object-oriented language: it is a lightweight mechanism that supports scalable extensibility, yet it is hardly noticeable to the novice programmer.

Many language extensions and design patterns have been proposed or implemented to address the limitations of inheritance, including virtual classes [20, 21, 33], mixins [2], mixin layers [31], delegation layers [29], higher-order hierarchies [10], and open classes [6]. A relationship between containment and inheritance is also introduced by virtual classes and higher-order hierarchies [10], but there are two key differences. First, unlike virtual classes, nested inheritance is statically type-safe; no run-time type checking is required to implement it. Second, nested inheritance associates nested classes with their containing classes rather than with objects of those classes.

The rest of this paper explores nested inheritance in more depth. Section 2 discusses why existing language mechanisms do not solve the problems that nested inheritance addresses. Section 3 presents nested inheritance. Section 4 describes the design of Jx and discusses adding nested inheritance to Java. We have implemented a prototype Jx compiler, described in Section 5. Because Jx is complex, a simpler language that captures the essence of nested inheritance is presented in Section 6, including its formal semantics and static type safety results. Section 7 discusses more broadly related work, and Section 8 concludes.

2 Scalable extensibility

Various programming language features support code reuse, including inheritance, parametric polymorphism, and mixins.

This technical report expands on the paper of the same name appearing in OOPSLA 2004. The only significant difference is the inclusion of the appendix containing proofs of soundness, starting on page 19.

But when code is reused, the programmer often finds that extension is not scalable: the amount of new code needed to obtain the desired changes in behavior is disproportionate to the perceived degree of change. More expressive language mechanisms are needed to make extension scalable.

2.1 Procedures vs. types

One reason why extension is often not scalable is the well-known difficulty of extending both types and the procedures that manipulate them [30, 36]. Object-oriented languages make it easy to add new types but not new procedures (methods) that operate on them; functional programming style makes it easy to add new procedures but not new types.

Extensions to an existing body of code are often *sparse* in the sense that new types that are added can be treated in a boilerplate way by most procedures, and the new procedures that are added have interesting behavior for only a few of the types on which they operate. However, standard programming methods cannot exploit this sparsity. In an object-oriented style, it is easy to add new classes, but to add new methods it is necessary to modify existing code, often duplicating the boilerplate code. In typical functional style, adding new functions that manipulate data is straightforward (assuming that the data representation is not encapsulated behind a module boundary), but modifying existing functions to handle new data types again requires modifying existing code.

This conflict is particularly noticeable in the context of an extensible compiler, where new types are added in the form of new abstract syntax nodes, and new procedures are added in the form of new compiler passes. With the usual strategy for compiler implementation, adding new abstract syntax requires changes to all passes, even if the new node types are relevant to only a few passes. Similarly, adding a new pass may require changes to all nodes, even if the pass interacts in an interesting way with only a few node types. Thus, the conflict between extending procedures and types creates an incentive to structure a compiler as a few complex passes rather than as a larger number of simple passes, resulting in a less modular compiler that is harder to understand, maintain, and reuse. Similar problems arise in other application domains, such as user interface toolkits.

Inheritance is a useful mechanism for extensibility because adding new types becomes more scalable: in general, a new type can inherit default behavior from some existing, similar type. However, inheritance does not handle extensions that need to add new fields or methods to an existing inheritance hierarchy in a uniform way. Some existing language mechanisms do help [6, 31, 29] but they do not solve the extensibility problems that we have encountered in developing Polyglot.

2.2 Hooks and extensibility

Making code extensible requires careful design so that the extension implementer has available the right hooks: interposition points at which new behavior or state can be added. However, there is often a price to pay: these hooks can often clutter or obfuscate the base code. One way to provide hooks is through language mechanisms that provide some kind of parametric genericity, such as parameterized types [19], parameterized mixins [2], and functors [23]. Explicit parameterization over types, classes, or modules precisely describes the ways in which extension is permitted. However, it is often an awkward way to achieve extensibility, especially when a number of modules are designed in conjunction with one another and have mutual dependencies. It is often difficult to decide which explicit parameters to introduce for purposes of future extension, and the overhead of declaring and using parameters can be awkward.

Inheritance embodies a different approach to extensibility. By giving names to methods, the programmer creates less obtrusive, *implicit* parameters that can be overridden when the code is reused. Nested inheritance builds on this insight by enabling nested classes to be used as hooks too.

3 Nested inheritance

Nested inheritance is a statically safe inheritance mechanism designed to be applicable to object-oriented languages that, like Java [13] or C++ [32], support nested classes or other containment mechanisms such as packages or namespaces. We have designed a language, Jx, that extends Java with nested inheritance. In this section, we concentrate on describing the nested inheritance mechanism, ignoring details of its interaction with Java and its implementation. These issues are discussed in Sections 4 and 5.

3.1 Overview

There are two key ideas behind nested inheritance. The first idea is similar to Ernst's higher-order hierarchies [10] and is related to virtual classes [20, 21]: a class inherits all members of its superclass—not only methods, but also nested classes and any subclass relationships among them.As with ordinary inheritance, the meaning of code inherited from the superclass is as if it were copied down from the superclass. A subclass may *override* any of the members it inherits. Like virtual classes, when a nested class is overridden, the overriding class does not replace the class it overrides, but instead *enhances* it. Thus, an overriding class is a subclass of the class it overrides, inheriting all its members. We extend this notion in one important way: the overriding class is not only a subclass but also a subtype of the class it overrides. This feature allows more opportunities for code reuse than with virtual classes or higher-order hierarchies. In addition, nested inheritance provides a form of *virtual superclasses* [21, 8], permitting the subclass relationships among the nested classes to be preserved when inherited into a new container class.[1] This feature allows new class members to be *mixed in* to a nested class by overriding its base class.

The second key idea in nested inheritance is a rich language for expressing types so that when code is inherited, types are reinterpreted in the context of the inheriting class. The innovation is an intuitive way to name types that gives the expressive power of virtual classes while also permitting sound typing.

Nested inheritance largely eliminates the need for factory methods [12] and other design patterns that address the problem of scalable extensibility [25]. Thus, a container such as a class or package may contain several nested classes or nested packages that depend on each other in complex ways. When the container is extended and individual components overridden, interactions between the components are preserved in the derived container.

The strength of nested inheritance as an extension mechanism is that it requires less advance planning to reuse code. Every class and method provides a hook for further extension,

[1] Note that the similar-sounding term "virtual base class" is used by C++ but has a very different meaning.

```
class A {
  class B { int x; }
  class C extends B {...}
  int m(B b) {
    return b.x;
  }
  C n() {
    return new C();
  }
}
```

```
class A2 extends A {
  class B { int y; }
  int m(B b) {
    return b.x + b.y;
  }
}
```

Figure 1: Nested inheritance example

so less programmer overhead is needed to identify the possible ways in which the code can be extended than in the functor and mixin approaches.

In this paper, nested inheritance is presented in the context of Java's nested classes. However, the same mechanism applies equally well to packages or other namespace abstractions. In the Jx language, packages may have a declared inheritance relationship; they act very much like classes whose components are all static. Section 3.7 discusses packages in more detail.

In Java, nested classes can be either inner classes or static nested classes. An instance of an inner class has a reference to an *enclosing instance* of its containing class; static nested classes do not have this pointer. This distinction is discussed further in Section 4.5. In the following discussion, we consider all nested classes to be static nested classes. This choice allows the mechanism to be applicable to classes nested within packages, which have no run-time instances.

3.2 A simple example

Consider the Java-like code in Figure 1. Because class `A` contains nested classes `B` and `C`, its subclass `A2` inherits nested classes `B` and `C` where the nested classes `A2.B` and `A2.C` are subclasses of `A.B` and `A.C`, respectively. Class `A2` explicitly declares a nested class `B`, overriding `A.B`; declarations within `A2.B` (such as the instance variable `y`) extend `A.B` as if `A2.B` were an explicitly declared subclass of `A.B`. Class `C` is inherited into `A2` as the *implicit class* `A2.C`. The programmer writes no code for `A2.C`; it is a subclass of both `A2.B` and `A.C`.

Subclass and subtype relationships are preserved by inheritance. For example, in Figure 1, the class `A2.C` is a subclass (and a subtype) of `A2.B` because `A.C` is a subclass of `A.B`. In addition, the constructor call `new C()` constructs an object of the class `A2.C` when the method `n` is invoked on an object of class `A2`.

Types named in inherited code are reinterpreted in the inheriting context. For example, the argument of the method `m` in the class `A` has type `B`, meaning `A.B` in the context of `A`. But when inherited into the class `A2`, the argument type becomes `A2.B` because the meaning of the name `B` is reinterpreted in the inheriting context. With this change, `A2` might not seem to conform to `A` because an argument method type has changed covariantly. However, subtyping between `A2` and `A` is still sound because the type system ensures the `m` method can only be called when its argument is known to be from the same implementation of `A` as the method receiver.

3.3 Compiler example

Figures 2 and 3 suggest how nested inheritance can be used to build an extensible compiler. Figure 2 gives simplified code

```
class Java {
  class Expr {
    Type type;
    void accept(Visitor v) {
      v.visitExpr(this);
    }
  }
  class Plus extends Expr {
    Expr left, right;
    void accept(Visitor v) {
      left.accept(v);
      right.accept(v);
      v.visitPlus(this);
    }
  }
  class Visitor {
    void visitExpr(Expr e) { }
    void visitPlus(Plus b) { }
  }
  class TypeChecker extends Visitor {
    void visitPlus(Plus p) {
      if (...) { p.type = Int; } else ...
    }
  }
}
```

Figure 2: Base compiler code

```
class Jif extends Java {
  class Expr { Label lbl; }
  class Label extends Expr { ... }
  class Visitor {
    void visitLabel(Label l) { }
  }
  class TypeChecker extends Visitor {
    void visitPlus(Plus p) {
      super.visitPlus(p);
      p.lbl = p.left.lbl.join(p.right.lbl);
    }
  }
}
```

Figure 3: Jif extension

for an ordinary Java compiler. Figure 3 uses nested inheritance to create a compiler for a language like Jif [24] that extends Java with information flow labels. This code uses the visitor pattern [12], in which compiler passes such as type checking are factored out into separate visitor objects, and boilerplate tree traversal is found in `accept` methods. The `Expr` and `Plus` classes implement abstract syntax tree (AST) nodes, and `TypeChecker` implements the type-checking pass, inheriting common functionality from its superclass `Visitor`.

Nested inheritance is effective for building this kind of extensible system. By adding a field `lbl` to the class `Expr`, every kind of expression node, including `Plus`, acquires this field. Similarly, adding a `visitLabel` method to `Visitor` causes every visitor, such as `TypeChecker`, to acquire this new method. The method `TypeChecker.visitPlus` can be then overridden to perform additional static checking on labels in addition to the ordinary type checking it performs by delegating to the superclass `Java.TypeChecker`. Note that the overridden `visitPlus` method expects a `Jif.Plus`, which has a `lbl` field, rather than a `Java.Plus`, which does not.

```
class A {
  class B {...}
  class C extends This.B {...}
  int m(this.class.B b) {
    return b.x;
  }
  this.class.C n() {
    return new this.class.C();
  }
}
```

Figure 4: Desugared version of class A from Figure 1

This example is suggestive of how nested inheritance could be used to implement the actual Polyglot and Jif compilers. Note that Jif.Expr and Java.Expr are different classes and both classes can coexist within the same compiler, permitting Jif abstract syntax trees to be translated to Java ASTs.

3.4 Naming types

The examples in Figures 1–3 look very much like Java; a Java programmer could be excused for not noticing the discrepancies. In fact, Jx is mostly backward compatible with Java: a Java program is a valid Jx program as long as nested classes are declared final or their containing classes are not subclassed. However, Jx obtains additional expressive power from new syntax for naming types (which is not shown in Figures 1–3). This syntax can be seen in Figure 4, which shows the class A from Figure 1 in a desugared form.

Class A.C is declared to extend This.B. When This is used in a declaration, it refers to the most specific class that inherits that declaration. In the body of A, This resolves to A and This.B therefore resolves to A.B. When C is inherited into A2, This.B is reinterpreted in the context of A2 and resolves to A2.B. Thus, A.C is a subclass of A.B and A2.C is a subclass of A2.B.

Returning to Figure 1, observe that the method m takes a formal parameter of type B. Since A2.B is a subclass of A.B, one might try to write unsafe code like the following, which passes an A.B to the method A2.m:

```
A a = new A2();
A.B b = new A.B();
a.m(b);
```

Because A.B does not have a y field, the behavior of the memory access b.y in the method m would be undefined. For this reason the above code does not type-check in Jx. Of course, this potential unsoundness results because the formal argument type is changed covariantly in the subclass A2. The virtual class mechanism in Beta [20] is unsound for precisely this reason, and therefore Beta requires a run-time check at method invocation. These checks create run-time overhead, but more importantly, they can lead to unexpected run-time errors. Our approach is instead to introduce a dependent type mechanism that ensures programs are statically safe and thus do not need run-time checks.

In Figure 1, the method A.m is declared with a formal parameter of type B, which is syntactic sugar for the type this.class.B, as shown in Figure 4. The *dependent class* this.class denotes the run-time class of the expression this, but *not* any subclass of the run-time class of this. As with ordinary non-dependent classes, a nested class can be selected from this.class. If the run-time class of this is A2, then this.class.B is really the class A2.B. If, at run time, this is an instance of class A, then this.class.B is A.B, but *not* A2.B.

Declaring the method parameter for m as this.class.B ensures that m in A2.B cannot be called with a superclass of A2.B. Callers of m must demonstrate that the method is invoked with a B selected from the receiver's class. In the following (safe) code, the variable a contains a value with run-time class A2.

```
final A a = new A2();
final a.class.B b = new a.class.B();
a.m(b);
```

To call the method m with receiver a, the caller must pass an argument of type a.class.B. Even if the receiver has static type A2, it is illegal to invoke m with an A2.B, since the actual run-time class of the receiver may be a subtype of A2 that overrides A2.m. The argument must have the type a.class.B. Note that a must be declared final to ensure its run-time class does not change.

In general, a dependent class is of the form p.class, where p is a final access path: either a final local variable (including formal parameters and this) or a field access p'.f, where p' is a final access path and f is a final field. The run-time class of an object specified by a final access path does not change.

The dependent type this.class is similar to the MyType (self type) construct of LOOM [3] and PolyTOIL [5]. The key difference is that with MyType, an instance of a subtype of MyType may be assigned to a variable of type MyType. Although MyType is covariant with respect to the subclassing relationship, the type MyType may be used as a method parameter type because subtyping and subclassing are decoupled. The dependent class p.class is also closely related to the path dependent type p.type in the ν*Obj* calculus [27] and in the Scala [26]; however p.type is a *singleton* type, meaning the only member of the type is the object referenced by p. p.class is not a singleton. In particular, one can create new instances of the class through the new operator (e.g., new p.class(...)).

While subclasses of the static type of a path a are not subtypes of a.class, the same is not true of classes selected relative to a.class. In particular, using the classes in Figure 1, a.class.C is a subtype of a.class.B, and therefore the call a.m(b) above is permitted.

3.5 Prefix types

Now consider the code in Figure 2, in which the classes Expr and Visitor are mutually recursive because of their respective accept and visitExpr methods. The class Jif extends Java, overriding both classes, so Jif.Expr and Jif.Visitor are mutually dependent in the same way as Java.Expr and Java.Visitor.

For code reuse, Expr and Visitor need to be able refer to each other without hard-coding the name of their enclosing class Java. Our solution is a type system that gives the ability to name the enclosing class of a given value.

For a non-dependent class P, and arbitrary class T, the *prefix type* $P[T]$ is the innermost enclosing class of T that is a subclass of P. Prefix types permit an unambiguous way of naming containers. For example, assuming the variable b has the static type A.B, then A[b.class] is the container of the run-time class of the value in b; if b contains a value of run-time class A2.B, then A[b.class] is the class A2.

In Figure 2 the method `Expr.accept` has a parameter with the (desugared) prefix type `Java[this.class].Visitor`, and `Visitor.visitExpr` has a parameter with the prefix type `Java[this.class].Expr`. When `accept` is invoked on a `Java.Expr`, it expects an argument of type `Java.Visitor`, but when invoked on a `Jif.Expr`, it expects `Jif.Visitor`. Thus, the relationship among the component classes is preserved. References to `Expr` within `Visitor` in Figure 2 are merely sugar for `Java[this.class].Expr`, and conversely for references to `Visitor` within `Expr`. No instance of the class `Java` need be in scope to use the type `Java[this.class].Expr`. This syntax thus makes it possible to refer to other classes in the current package even though packages do not have instances.

3.6 Overriding the superclass

When overriding a class in a containing class, the programmer can change the superclass. This feature allows new functionality to be mixed in to several classes in the new containing class without code duplication.

The superclass of a nested class *bounds* the type of the nested class. Overriding the superclass permits this bound to be tightened, enabling a virtual type-like pattern. In particular, if `D` is a nested class that extends some other class `C`, then `D` is like a virtual type, bounded by `C`; when `D`'s container is subclassed, the superclass of `D` can be modified to be a subclass of the original superclass of `D`. This has the effect of making the virtual type `D` more precise in the container's subclass.

3.7 Package inheritance

The language mechanisms described for nested inheritance apply to packages as well as to classes. Indeed, we expect nested inheritance of packages to be the most common use of nested inheritance.

In Jx, packages, like classes, may have a declared inheritance relationship. If package `P2` extends package `P`, then `P2` inherits all members of package `P`, including nested packages.[2] The declaration that `P2` extends `P` is made in a special source file in the package `P2`, which facilitates separate compilation by allowing the package `P` to be ignorant of its descendants. The declaration is *not* made in each separate source file of the package `P2`, since doing so would duplicate package inheritance declarations, introducing possible inconsistencies and making modification of the inheritance relationship more difficult.

Prefix types extend to accommodate packages: if P is a package name and T is an arbitrary class, then P`[`T`]` is the innermost enclosing package of T that is derived from P. Prefix types may also appear in `import` declarations. For example, consider a package `P` with nested packages `Q` and `R`, and a source file in `Q` that imports classes from `R`. To allow code reuse via nested inheritance, these classes must be imported without hard-coding the names of their enclosing packages. The source file in `Q` uses the declaration `import P[Package].R.*` to import the appropriate classes. The keyword `Package` refers to the package of the most specific class that inherits the import declaration, analogous to the use of `This` in a declaration to denote the most specific class that inherits that declaration. We use the name `Package` since neither `This` nor `this` are in scope at import declarations.

[2] Nested packages are called *subpackages* in Java [13]. We refrain from using this term to avoid confusion between nested packages and derived packages.

Dependent classes, on the other hand, do not need to be extended to handle packages because packages do not have run-time instances.

3.8 Genericity

Nested inheritance is intended to be a mechanism for extensibility and not for genericity. Jx is an extension of Java and, as of version 1.5, Java already has a genericity mechanism, parameterized types.

Nested inheritance as presented above does not provide an abstract type construct. To use virtual types for genericity, abstract types are used to equate a virtual type with a class. For example, the following code fragment implements a generic `List` class and a `List` of `Integers`, `IntList`, in a hypothetical extension of Jx with abstract types.

```
class List {
  abstract class T extends Object { }
  void add(this.class.T x) { ... }
}
class IntList extends List {
  class T = Integer;
}
```

By declaring `IntList.T` to be an alias for `Integer`, the `add` method may be called with an argument of type `Integer`. Without abstract types, the best that can be done using nested classes is to declare `IntList.T` as

```
class T extends Integer { }
```

But in this case, only instances of `IntList.T` can be added to an `IntList`, not instances of the `Integer` class. However, a list of `Integer` can be implemented more succinctly as the parameterized type `List<Integer>`.

3.9 Final binding

As in Java, classes in Jx may be declared `final` to prevent the class from being subclassed. This naturally extends to nested inheritance be requiring that a `final` nested class can be neither subclassed explicitly with an `extends` declaration nor overridden in a subclass of its enclosing class. This *final binding* of nested classes is useful for enabling optimizations and for modeling purposes. In addition, virtual classes in Beta may be inherited from only if they are final bound. Jx does not permit inheritance from dependent classes and thus this restriction is not needed.

Final classes also enable backward compatibility with Java; if all nested classes are `final`, a Jx program is a legal Java program.

4 Interactions with Java

Nested inheritance introduces several new features that are discussed in Section 3. It is worth discussing how these features interact with some existing object-oriented programming features in Java.

4.1 Conformance

In Jx, a class conforms to its superclass under the same rules as in Java: a method's parameter types and return type must be identical in both classes. In principle this rule could be relaxed to permit covariant refinement of method return types, but we have not explored this relaxation.

```
class A {
 class B {
  int m() { return 0; }
 }
 class B2 extends B {
  int m() { return 1; }
 }
}
class A2 extends A {
 class B {
  int m() { return 2; } }
 }
}
```

(a) Original code

```
class A2 extends A {
 class B_inh {
  int m() { return 0; }
 }
 class B extends B_inh {
  int m() { return 2; }
 }
 class B2_inh extends B {
  int m() { return 1; }
 }
}
```

(b) A2 with implicit classes shown in *italics*

Figure 5: Method dispatch example

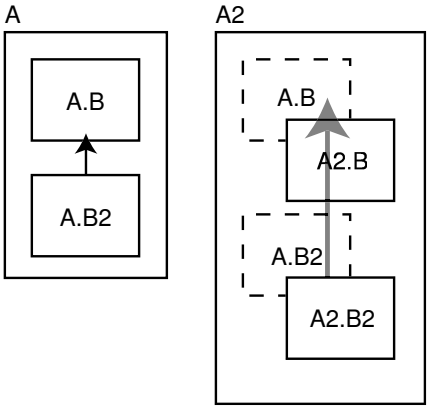

Figure 6: Dispatch order

4.2 Method dispatch

In Java, method calls are dispatched to the method body in the most specific class of the receiver that implements the method. In the code in Figure 5(a), both `A2.B` and `A.B2` override `A.B`'s implementation of `m`. The implicit class `A2.B2` inherits `m` from both `A.B2` and `A2.B`. Which of the two implementations is the most specific?

The same issue arises in languages that support multiple inheritance. For example, in C++ this situation is considered an error. However, because nested inheritance introduces implicit classes, this rule would effectively prevent a class from overriding any methods of a class it overrides, since its implicit subclasses would inherit both implementations.

Instead, we exploit the structure of the inheritance mechanism. When `A` is subclassed to `A2`, if `B` is not overridden, it is an implicit class of `A2`. We write this class $A2.B_{inh}$. Now when `A2.B` is declared, overriding `A.B`, we can consider its immediate superclass to be *not* `A.B`, but rather the implicit class $A2.B_{inh}$ inherited into `A2`. We can think of the code for `A2` in Figure 5(a) as the code in Figure 5(b). Thus, in order from most to least specific, the classes in `A2` are: $A2.B2_{inh}$, `A2.B`, and $A2.B_{inh}$, or equivalently: `A.B2`, `A2.B`, and `A.B`. This dispatch order is depicted in Figure 6.

This dispatch order is not chosen arbitrarily: `A.B2` should be dispatched to before `A2.B` because the `B2` classes are specializations of the `B` classes, and thus all `B2` classes should be regarded as being more specific than any `B` class. The same dispatch order is used in delegation layers [29].

```
class A {
  class B { }
  class B2 extends B {
    int m() {...}
  }
}
```

```
class A2 extends A {
 class B {
  Object m() {...}
 }
 class B2 extends B {
  void n() {
   m(); // A.B2.m() or
        // A2.B.m()?!
  }
 }
}
```

Figure 7: Name conflict example

4.3 Naming conflicts

To support separate compilation of classes, Jx needs a mechanism for resolving naming conflicts, which arise when a class inherits more than one implementation of a given method or field. For example, consider the code in Figure 7. The classes `A.B2` and `A2.B` have a common ancestor `A.B`, and both declare a method `m()`, but with incompatible return types. Both of these method declarations are allowed, because in general, each class could be compiled independently of the other—particularly, if the container `A` were a package instead of a class. However, in the method body of `A2.B2.n()`, it is not clear which method `m()` is referred to. In addition, if `A2.B2` wished to override one or both of the methods `m()`, then the method declarations need to indicate which method they are overriding.

Jx resolves naming conflicts for calls by requiring the caller to cast the receiver of the method invocation to a class in which there is no such conflict. For example, in `A2.B2.n()`, the method call `((A2.B)this).m()` would be permitted, as the name `m()` is not in conflict in the class `A2.B`. Field accesses are handled similarly.

Naming conflicts for method overriding are resolved by ensuring the overriding method declaration supplies the class name of an ancestor class on which the overridden method is defined. For example, if the class `A2.B2` wished to override the method `m()` declared in class `A.B2`, the method declaration in `A2.B2` would be written `int A.B2.m() {...}`.

Since we expect naming conflicts to be exceptional, rather than the norm, the additional mechanisms required by Jx to resolve naming conflicts should not be overly burdensome.

4.4 Constructors

Nested inheritance requires that constructors, like methods, are inherited by subclasses, so that it is possible to call constructors of dependent classes and prefix types. Suppose that the class `A.B` contains a constructor that takes an integer as an argument. Then the following code is permitted:

```
final A a = new A2();
final a.class.B b = new a.class.B(7);
```

The expression `new a.class.B(7)` is allowed because the statically known type of `a` is the class `A`, and there is a suitable constructor for the class `A.B`. However, at runtime the variable `a` contains a value of run-time class `A2`, and therefore an object of class `A2.B` is constructed. In order to be sound, the

class A2.B must have a constructor with a suitable signature. Since A2.B may in general be an implicit class, A2.B must inherit the constructors of A.B, and of any other superclasses, in the same way that it inherits methods.

The primary use of constructors is for initializing fields; if a final field does not have an initializer, then every constructor of the class must ensure that the final field is initialized. Initializing final fields is particularly important for nested inheritance, because some final fields may be used to define dependent classes. Failure to initialize these fields would lead to unsoundness. Therefore, if a class declares a final field, that field must either have an initializer, or else all constructors inherited from superclasses must be overridden and that field must be initialized in each constructor.

4.5 Inner classes

We have assumed that nested classes are static and are thus not inner classes. An instance of a static nested class does not have a reference to an enclosing instance of its container class. In Java, these enclosing instances are written P.this, where P is the name of an enclosing class. Jx can accommodate inner classes by assigning the type P[this.class] to the enclosing instance P.this.

Allowing inner classes raises the possibility of extending Jx to allow dependent classes to appear in the extends clause of nested classes. For example, if the class A had inner class B and a final field f, then B could be declared to extend this.f.class. Dependent classes cannot currently appear in the extends clause of a nested class, as this is not in scope during the declaration of a static nested class.

If the use of dependent classes in extends clauses is restricted to this.class or prefixes of this.class, then the current type system of Jx suffices, because this.class is equivalent to This when this is in scope. References to enclosing instances can be implemented as fields of the nested instance, as is done by javac and by Igarashi and Pierce's formalization of inner classes [16]. However, if arbitrary dependent classes are allowed, such as this.f.class, then the type system of Jx would need to be modified, and the implementation described later, in Section 5, would need significant redesign.

5 Implementation

We have implemented a prototype translation from Jx to Java as a 3700-line extension in the Polyglot compiler framework [25]. The prototype supports class inheritance but not package inheritance as described in Section 3.7. However, a design for implementing package inheritance is presented in Section 5.4. The translation is efficient in that it does not duplicate code, although each Jx class, including implicit member classes, is represented explicitly in the target language.

5.1 Translating classes

As depicted in Figure 8, each source Jx class (including implicit member classes) is represented in translation by two Java classes and two Java interfaces: the *instance class*, the *method interface*, the *class class*, and the *static interface*.

The *instance class* for a Jx class C contains the translation of any methods and constructors declared in C. An object of the Jx class C is represented at runtime by a collection of instance class objects, one instance class object for C and each Jx class that C subclasses. The instance objects that represent C point

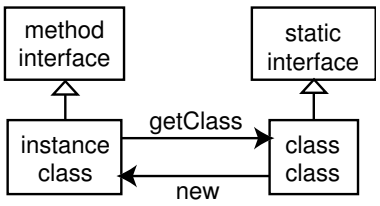

Figure 8: Target classes and interfaces

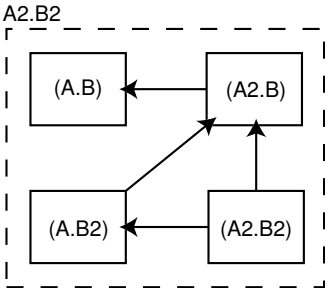

Figure 9: Representation of an A2.B2 object

to each other via *dispatch fields*. For example, the class A2.B2 of Figure 5 is represented by four objects as shown in Figure 9. The instance class also provides methods for accessing fields and for dispatching to methods, including those C inherits; these dispatch methods simply forward the field access or method call to an appropriate instance object of a superclass of C, using the dispatch fields. Note that Java's normal method dispatch mechanism cannot be used, because instance objects of superclasses of C are not superclasses of C's instance object. Hence, the translation must make dispatch explicit.

Each instance class has two constructors: a *master* constructor and a *slave* constructor. If an object of class C is being created, then the master constructor of C's instance class is invoked, creating the other instance objects needed to represent a Jx C object by invoking the necessary slave constructors. The slave constructor of C's instance class is invoked when the instance object is being used to represent a subclass of C.

The instance class also contains the translations of the Jx constructors of C. Jx constructors are translated into methods in the instance class, which are invoked by the class class (see below); the translation of constructors into methods facilitates the inheritance of constructors.

The instance class for C implements the *method interface* for C, which declares all methods that C defines, as well as getter and setter methods for all non-private fields declared in C. The method interface extends all the method interfaces of C's superclasses.

The *class class* provides means at runtime to both access type information about C and create new C objects (that is, collections of appropriate instance classes). For every Jx class, there is a single class class object instantiated at runtime. Every instance class has a method that returns the appropriate class class, analogous to Java's getClass() method on the Object class.

Information about C's superclasses, enclosing class, and nested classes is available at runtime in order to create instances of prefix types. For example, if v is a Jx object, and a new object of type P[v.class] needs to be created via a constructor call new P[v.class](...), then v's class class

must be interrogated to find the class class for the most specific enclosing class of `v.class` that is a subclass of `P`. The class class object found is then used to create the new object: the class class for `C` has a method `newThis(...)` for every constructor declared or inherited by `C`. These methods create a new instance class object for `C`, with the master constructor, and then invoke the appropriate translated constructor on the instance class object.

The class class also provides a method to test if a given object is an instance of the Jx class, and a `cast(Object o)` method, which throws a `ClassCastException` if the object `o` is not an instance of the Jx class, and returns `o` otherwise. These methods are needed to support the translation of casts and `instanceof` expressions in the source language.

The class class implements the *static interface*, which declares all constructors that `C` declares or inherits. The static interface extends all static interfaces of `C`'s superclasses.

All methods on class class objects are invoked via an appropriate static interface. This permits the translation of constructor calls on dependent classes. For example, suppose `A2` is a subclass of `A`. Then `A2`'s class class implements `A`'s static interface. Now, if the variable `a` has static type `A`, the Jx expression `new a.class()` will be translated to a call to `newThis()` on `A`'s static interface. Supposing that the runtime class of `a` is `A2`, then that method call will actually invoke `newThis()` on `A2`'s class class, and thus create a new instance of `A2`.

5.2 Translating methods

A method declaration in a Jx class `C` is translated into a method declaration in `C`'s instance class; any method that `C` inherits has a dispatch method created in `C`'s instance class.

Since a Jx object is represented at runtime by a collection of instance objects, the source language expression `this` must be translated into something other than the target language expression `this`, in order to allow method invocations and field accesses on the Jx object. To achieve this, the translation adds an additional parameter `self` to every source language method and constructor. The `self` parameter is the translation of the special variable `this` and always refers to the master instance object, the instance object that created the other instance objects that collectively represent a Jx object.

5.3 Translating fields

A field declaration in a Jx class `C` is translated into a field declaration in `C`'s instance class. Getter and setter methods are also produced for any non-private fields, which allows the method dispatch mechanism to be used to access the fields. Field accesses in Jx code are translated into calls to the getter and setter methods.

5.4 Translating packages

This section describes a design for translating package inheritance in Jx. This design is not yet implemented.

Packages, like classes, require a means to access type information about the package at runtime. For a given package `P`, the *package class* for `P` provides type information about `P` to resolve prefix types, analogous to a class class. The package class is able to provide information about what package `P` inherits from, the package that contains `P`, packages nested inside `P`, and classes contained in the package `P`.

Since a package class needs to know about all classes in the package, care must be taken to ensure that the classes in a given package can be compiled separately while guaranteeing that the package class contains correct information. Correctness can be achieved by generating the package class every time a class within the package is compiled, under the assumption that all previously compiled classes within the package are available at that time. Removal of a class from a package requires the package class to be regenerated. The reflection mechanism of Java may provide a more flexible mechanism to ensure the correctness of information provided by package classes.

6 Simple language model

To explore the soundness of type checking with nested inheritance, we developed a simple Java-like language that demonstrates the core features of nested inheritance with dependent classes. For simplicity, many features of the full Jx language are absent. In particular, the language presented here includes nested classes but not packages. A package can be modeled as a class in which all classes in the package are nested.

The language is based on Featherweight Java (FJ) [15], but includes a number of additional features found in the full Java language—notably, a heap and `super` calls—needed to model important features of nested inheritance. We include a heap in order to model recursive data structures, which interact with dependent classes in non-trivial ways. The language includes static nested classes, dependent classes and prefix types.

6.1 Syntax

The syntax of the language is shown in Figure 10. We write $\vec{x}$ to mean the list $x_1, \ldots, x_n$ and $\overline{x}$ to mean the set $\{x_1, \ldots, x_n\}$ for some $n \geq 0$. A term with list subterms (e.g., $\vec{f} = \vec{e}$) should be interpreted as a list of those terms (i.e., $f_1 = e_1, \ldots, f_n = e_n$). We write $\#(\vec{x})$ for the length of $\vec{x}$. The empty list is written $[]$. The singleton list containing x is denoted $[x]$. We write $x, \vec{x}$ for the list with head x and tail $\vec{x}$, and $\vec{x}_1 \,,\, \vec{x}_2$ for the concatenation of $\vec{x}_1$ and $\vec{x}_2$.

A program *Pr* is a pair $\langle \overline{L}, e \rangle$ of a set of top-level class declarations L and an expression e, which models the program's `main` method. To simplify presentation, we assume a single global *top-level class table TCT*, which maps top-level class names C to their corresponding class declarations $\texttt{class}\ C\ \texttt{extends}\ S\ \{\overline{L}\ \vec{F}\ \overline{M}\}$.

A class declaration L may include a set of nested class declarations $\overline{L}$, a list of fields $\vec{F}$, and a set of methods $\overline{M}$. Fields are in a list since the order of the fields is important for field initialization. There are two forms of class declaration L. In the *TCT*, a class declaration's `extends` clause cannot mention a dependent class, but it may refer to the *type schema* `This`, which is used to name the enclosing class into which the class declaration is inherited. During class lookup, `This` is replaced with the name of the enclosing class, producing a class declaration with an `extends` clause of the form `extends` T.

Types T are either top-level classes C, qualified types $T.C$, dependent classes $p.\texttt{class}$, or prefix types $P[T : P.C]$, where P denotes a non-dependent class name. A type may depend on an access path expression p; the dependent class $p.\texttt{class}$ is the run-time class of the object referred to by access path p. To be a well-formed type, p must be a `final` access path; if p were mutable, the class of the object it refers to could change at run time, leading to an unsoundness. A prefix type $P[T : P.C]$ is the innermost enclosing class T' of T such that T' is a subtype of P and T is a subtype of $T'.C$ (and thus of $P.C$). For the prefix type to be well-formed $P.C$ must exist

Syntax:

programs	Pr	$::= \langle \overline{L}, e \rangle$
class declarations	L	$::= \texttt{class}\ C\ \texttt{extends}\ S\ \{\overline{L}\ \vec{F}\ \overline{M}\}$
		$\mid \texttt{class}\ C\ \texttt{extends}\ T\ \{\overline{L}\ \vec{F}\ \overline{M}\}$
type schemas	S	$::= C \mid S.C \mid \texttt{This} \mid P\texttt{[}S\texttt{:}P.C\texttt{]}$
types	T	$::= C \mid T.C \mid p.\texttt{class}$
		$\mid P\texttt{[}T\texttt{:}P.C\texttt{]}$
simple nested classes	P, Q	$::= C \mid P.C$
field declarations	F	$::= [\texttt{final}]\ T\ f = e$
method declarations	M	$::= T\ m(\vec{T}\ \vec{x})\ \{e\}$
access paths	p	$::= v \mid p.f$
base values	b	$::= \ell_P \mid \texttt{null}$
values	v	$::= b \mid x$
expressions	e	$::= p$
		$\mid \texttt{final}\ T\ x = e_1;\ e_2$
		$\mid p.f =_{[\texttt{final}]} e_1;\ e_2$
		$\mid p.m(\vec{v})$
		$\mid v.\texttt{super}_P.m(\vec{v})$
		$\mid \texttt{new}\ T\ \texttt{as}\ x\ \{\vec{f} = \vec{e}\}$
objects	o	$::= P\ \{\overline{f} = \overline{\ell_P}\}$
typing environments	Γ	$::= \emptyset \mid \Gamma, x : T$

Evaluation contexts:

evaluation contexts	E	$::= [\cdot]$
		$\mid \texttt{final}\ TE\ x = e_1;\ e_2$
		$\mid \texttt{final}\ T\ x = E;\ e$
		$\mid E.f$
		$\mid E.f = e_1;\ e_2$
		$\mid b.f = E;\ e_2$
		$\mid E.m(\vec{b})$
		$\mid \texttt{new}\ TE\ \texttt{as}\ x\ \{\vec{f} = \vec{e}\}$
type eval contexts	TE	$::= TE.C$
		$\mid P\texttt{[}TE\texttt{:}P.C\texttt{]}$
		$\mid E.\texttt{class}$
null eval contexts	N	$::= \texttt{null}.f$
		$\mid \texttt{final}\ TE[\texttt{null}]\ x = e_1;\ e_2$
		$\mid \texttt{null}.f = b;\ e$
		$\mid \texttt{null}.m(\vec{b})$
		$\mid \texttt{null}.\texttt{super}_P.m(\vec{b})$
		$\mid \texttt{new}\ TE[\texttt{null}]\ \texttt{as}\ x\ \{\vec{f} = \vec{e}\}$

Type interpretation:

$$\begin{aligned}
\mathit{exact\text{-}class}(\ell_P.\texttt{class}) &= P \\
\mathit{exact\text{-}class}(P\texttt{[}T\texttt{:}P.C\texttt{]}) &= \mathit{prefix}(P, \mathit{exact\text{-}class}(T), \\
&\qquad \mathit{exact\text{-}class}(T), P.C) \\
\mathit{runtime\text{-}class}(C) &= C \\
\mathit{runtime\text{-}class}(T.C) &= \mathit{runtime\text{-}class}(T).C \\
\mathit{runtime\text{-}class}(\ell_P.\texttt{class}) &= P \\
\mathit{runtime\text{-}class}(P\texttt{[}T\texttt{:}P.C\texttt{]}) &= \mathit{prefix}(P, \mathit{runtime\text{-}class}(T), \\
&\qquad \mathit{runtime\text{-}class}(T), P.C) \\
\mathit{prefix}(P, P_0, P'.C, P.C) &= P' \\
\mathit{prefix}(P, P_0, T, P.C) &= \mathit{prefix}(P, P_0, (\emptyset, P_0, T), P.C) \\
&\qquad (T \neq P'.C \text{ for any } P')
\end{aligned}$$

Class lookup:

$$\frac{\mathit{classes}(\Gamma, T_0, P) = \overline{L_s} \quad \mathit{TCT}(C) = C\ \texttt{ext}\ P\ \{\overline{L}\ \vec{F}\ \overline{M}\}}{\mathit{CT}(\Gamma, T_0, C) = C\ \texttt{ext}\ P\ \{\overline{L_s} \bullet \overline{L}\{T_0/\texttt{This}\}\ \vec{F}\ \overline{M}\}} \quad \text{(CT-OUTER)}$$

$$\frac{C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\} \in \mathit{classes}(\Gamma, T, T) \quad \mathit{classes}(\Gamma, T_0, T_s) = \overline{L_s}}{\mathit{CT}(\Gamma, T_0, T.C) = C\ \texttt{ext}\ T_s\ \{\overline{L_s} \bullet \overline{L}\{T_0/\texttt{This}\}\ \vec{F}\ \overline{M}\}} \quad \text{(CT-NEST)}$$

$$\frac{\mathit{exact\text{-}class}(T) = P \quad \mathit{classes}(\Gamma, T_0, P) = \overline{L}}{\mathit{CT}(\Gamma, T_0, T) = \_\ \texttt{ext}\ P\ \{\overline{L} \bullet \emptyset\}} \quad \text{(CT-RUNTIME)}$$

$$\frac{P\texttt{[}T\texttt{:}P.C\texttt{]} \notin \mathrm{dom}(\mathit{exact\text{-}class}) \quad \mathit{classes}(\Gamma, T_0, P) = \overline{L}}{\mathit{CT}(\Gamma, T_0, P\texttt{[}T\texttt{:}P.C\texttt{]}) = \_\ \texttt{ext}\ P\ \{\overline{L} \bullet \emptyset\}} \quad \text{(CT-PRE)}$$

$$\frac{p.\texttt{class} \notin \mathrm{dom}(\mathit{exact\text{-}class}) \quad \Gamma \vdash p\ \texttt{final}\ P \quad \mathit{classes}(\Gamma, T_0, P) = \overline{L}}{\mathit{CT}(\Gamma, T_0, p.\texttt{class}) = \_\ \texttt{ext}\ P\ \{\overline{L} \bullet \emptyset\}} \quad \text{(CT-DEP)}$$

Member class inheritance:

$$\overline{L_1} \bullet \overline{L_2} = \bigcup_{C \in \mathrm{dom}(\overline{L_1} \cup \overline{L_2})} \overline{L_1}(C) \bullet \overline{L_2}(C)$$

$$\overline{L}(C_i) = \begin{cases} L_i & \text{if } L_i = C_i\ \texttt{ext}\ T_i\ \{\overline{L_i}\ \vec{F_i}\ \overline{M_i}\} \\ \mathit{absent} & \text{otherwise} \end{cases}$$

$$C\ \texttt{ext}\ T_1\ \{\overline{L_1}\ \vec{F_1}\ \overline{M_1}\} \bullet C\ \texttt{ext}\ T_2\ \{\overline{L_2}\ \vec{F_2}\ \overline{M_2}\} = C\ \texttt{ext}\ T_2\ \{\overline{L_1} \bullet \overline{L_2}\ \vec{F_2}\ \overline{M_2}\}$$

$$C\ \texttt{ext}\ T_1\ \{\overline{L_1}\ \vec{F_1}\ \overline{M_1}\} \bullet \mathit{absent} = C\ \texttt{ext}\ T_1\ \{\overline{L_1} \bullet \emptyset\}$$

$$\mathit{absent} \bullet C\ \texttt{ext}\ T_2\ \{\overline{L_2}\ \vec{F_2}\ \overline{M_2}\} = C\ \texttt{ext}\ T_2\ \{\overline{L_2}\ \vec{F_2}\ \overline{M_2}\}$$

Final access paths:

$$\frac{\vdash P\ \text{wf}}{\vdash \ell_P\ \texttt{final}\ P} \quad \text{(F-LOC)}$$

$$\frac{\Gamma \vdash T\ \text{wf}}{\Gamma \vdash \texttt{null}\ \texttt{final}\ T} \quad \text{(F-NULL)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x\ \texttt{final}\ T} \quad \text{(F-VAR)}$$

$$\frac{\Gamma \vdash p\ \texttt{final}\ T \quad \mathit{ftype}(\Gamma, T, f_i) = \texttt{final}\ T_i}{\Gamma \vdash p.f_i\ \texttt{final}\ T_i\{p/\texttt{this}\}} \quad \text{(F-GET)}$$

$$\frac{\Gamma \vdash p\ \texttt{final}\ T \quad \mathit{exact\text{-}class}(T) = P \quad \mathit{exact\text{-}class}(T') = P}{\Gamma \vdash p\ \texttt{final}\ T'} \quad \text{(F-RUNTIME)}$$

Figure 10: Syntax and class lookup functions

Superclasses:

$$\frac{CT(\Gamma,T,T) = C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}}{super(\Gamma,T) = T_s}$$

Nested classes:

$$classes(\Gamma,T_0,\texttt{Object}) = \emptyset$$

$$\frac{CT(\Gamma,T_0,T) = C\ \texttt{ext}\ T'\ \{\overline{L}\ \vec{F}\ \overline{M}\}}{classes(\Gamma,T_0,T) = \overline{L}}$$

Fields:

$$fields(\Gamma,T_0,\texttt{Object}) = [\,]$$

$$\frac{\begin{array}{c}CT(\Gamma,T_0,T) = C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}\\ (\Gamma,T_0,T) = T'\\ fields(\Gamma,T_0,T') = \vec{F}'\end{array}}{fields(\Gamma,T_0,T) = \vec{F}',\vec{F}}$$

$$\frac{fields(\Gamma,T,T) = [\texttt{final}]\ \vec{T}\ \ \vec{f} = \vec{e}}{ftype(\Gamma,T,f_i) = [\texttt{final}]\ T_i}$$

$$\frac{fields(\Gamma,T,T) = [\texttt{final}]\ \vec{T}\ \ \vec{f} = \vec{e}}{finit(\Gamma,T,f_i) = e_i}$$

$$\frac{fields(\Gamma,T,T) = [\texttt{final}]\ \vec{T}\ \ \vec{f} = \vec{e}}{fnames(\Gamma,T) = \overline{f}}$$

Methods:

$$\frac{\begin{array}{c}CT(\Gamma,T_0,T) = C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}\\ T_r\ m(\vec{T}\ \vec{x})\ \{e\} \in \overline{M}\end{array}}{method(\Gamma,T_0,T,m) = T_r\ m(\vec{T}\ \vec{x})\ \{e\}}$$

$$\frac{\begin{array}{c}CT(\Gamma,T_0,T) = C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}\\ T_r\ m(\vec{T}\ \vec{x})\ \{e\} \notin \overline{M}\\ (\Gamma,T_0,T) = T'\\ method(\Gamma,T_0,T',m) = M\end{array}}{method(\Gamma,T_0,T,m) = M}$$

$$\frac{method(\emptyset,T_0,T,m) = T_r\ m(\vec{T}\ \vec{x})\ \{e\}}{mbody(T_0,T,m) = (\vec{x},e)}$$

$$\frac{method(\Gamma,T_0,T,m) = T_r\ m(\vec{T}\ \vec{x})\ \{e\}}{mtype(\Gamma,T_0,T,m) = (\vec{x}:\vec{T}) \to T_r}$$

Operational semantics:

$$\frac{runtime\text{-}class(T) = P}{\langle H,\texttt{final}\ T\ x = b;\ e\rangle \longrightarrow \langle H,e\{b/x\}\rangle} \quad \text{(R-LET)}$$

$$\frac{H(\ell_P) = P\ \{\overline{f} = \overline{b}\}}{\langle H,\ell_P.f_i\rangle \longrightarrow \langle H,b_i\rangle} \quad \text{(R-GET)}$$

$$\frac{\begin{array}{c}H(\ell_P) = P\ \{\overline{f} = \overline{b}\}\\ H' = H[\ell_P := P\ \{f_1 = b_1,\ldots,f_i = b_i',\ldots,f_n = b_n\}]\end{array}}{\langle H,\ell_P.f_i =_{[\texttt{final}]} b_i';\ e\rangle \longrightarrow \langle H',e\rangle} \quad \text{(R-SET)}$$

$$\frac{mbody(P,P,m) = (\vec{x},e)}{\langle H,\ell_P.m(\vec{b})\rangle \longrightarrow \langle H,e\{\ell_P/\texttt{this},\vec{b}/\vec{x}\}\rangle} \quad \text{(R-CALL)}$$

$$\frac{(\emptyset,P,Q) = Q' \quad mbody(P,Q',m) = (\vec{x},e)}{\langle H,\ell_P.\texttt{super}_Q.m(\vec{b})\rangle \longrightarrow \langle H,e\{\ell_P/\texttt{this},\vec{b}/\vec{x}\}\rangle} \quad \text{(R-SUPER)}$$

$$\frac{\begin{array}{c}runtime\text{-}class(T) = P\\ fnames(\emptyset,P) = \vec{f}'\\ \overline{f} \subseteq \overline{f'}\\ \ell_P \notin \mathrm{dom}(H)\\ H' = H[\ell_P = P\ \{\overline{f'} = \texttt{null}\}]\\ e_i' = e_i\{\ell_P/x\} \text{ if } f_i \in \overline{f}\\ e_i' = finit(\emptyset,P,f_i)\{\ell_P/\texttt{this}\} \text{ if } f_i \in \overline{f'} - \overline{f}\\ e'' = \ell_P.\vec{f}' =_{\texttt{final}} \vec{e'};\ \ell_P\end{array}}{\langle H,\texttt{new}\ T\ \texttt{as}\ x\ \{\vec{f} = \vec{e}\}\rangle \longrightarrow \langle H',e''\rangle} \quad \text{(R-NEW)}$$

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

$$\langle H,E[N]\rangle \longrightarrow \langle H,\texttt{null}\rangle \quad \text{(R-NULL)}$$

Dispatch ordering:

$$\frac{ord(\Gamma,T) = \vec{T}}{(\Gamma,T,T_i) = T_{i+1}}$$

$$\begin{array}{c}ord(\Gamma,\texttt{Object}) = [\texttt{Object}]\\ ord(\Gamma,T.C) = ord(\Gamma,T).C,\ ord(\Gamma,super(\Gamma,T.C))\\ ord(\Gamma,T) = T,ord(\Gamma,super(\Gamma,T))\\ \text{where } T \neq \texttt{Object} \text{ and}\\ T \neq T'.C \text{ for any } T'\end{array}$$

$ord(\Gamma,T).C$ is the list of $T'.C$ such that $T' \in ord(\Gamma,T)$ and $\Gamma \vdash T'.C$ wf

Figure 11: Member lookup functions and operational semantics

and T must be a dependent class or another prefix type. This definition of prefix type differs from the description given in Section 3; the change simplifies the semantics. Although the prefix type syntax can name only the immediately enclosing class of T, further enclosing classes can be named by prefixing the prefix type (e.g., `A[A.B[x.class:A.B.C]:A.B]`).

Fields F may declared `final` or non-`final`. All field declarations include an initializer expression. The syntax for methods M is similar to that of Java.

Expressions e are similar to Java expressions of the same form. Access paths p are either field accesses $p.f$ or values v, which include base values b and variables x. Base values b are either memory locations ℓ_P of type P or `null`. Locations are not valid surface syntax, although they appear during evaluation. All variables x, including formal parameters and the special variable `this`, are `final` and are initialized at their declaration. The declaration `final` $T\ x = e_1;\ e_2$ initializes x to e_1, then evaluates e_2.

Fields and methods are accessed only through final access paths p. Field assignments may optionally be annotated with the keyword `final`, permitting assignment to `final` fields when initializing an object. These `final` assignments are not allowed in the surface syntax. Methods dispatch to the method body in the most specific superclass of the receiver, as described in Section 4.2. A method implemented by a superclass of P may be invoked with the expression $v.\texttt{super}_P.m(\vec{v})$. In the surface syntax, v must be `this`, but v can take on arbitrary values during evaluation as substitutions occur. To simplify dispatch, a `super` call is marked with the name of the class lexically P containing the call.

Allocation is performed with the `new` operator. The calculus does not include constructors. Instead, the `new` operator has an *inline constructor body* that may initialize zero or more fields of the new object. The field initializers may refer to the new object through the variable x. Fields not assigned in the inline constructor body are initialized with their default initializers. Field initialization order is left undefined; fields are initialized to `null` by default. Access to an uninitialized field is treated as a `null` dereference. A heap H maps locations ℓ_P to objects o, which are simple records annotated with their class type.

For any term t, value v, and variable x we write $t\{v/x\}$ for the capture-free substitution of v for x in t. As is standard practice, α-equivalent terms are identified. We write $FV(t)$ for the set of free variables in t.

6.2 Class lookup

Classes are defined in a fixed top-level class table *TCT* that maps all top-level class names C to class declarations L. We extend the top-level class table *TCT* to a function *CT*, shown in Figure 10. *CT* returns class declarations not only for top-level class names, but for arbitrary types. Member lookup and subtyping are defined using *CT*.

In addition to the type to lookup, *CT* has two more parameters. Because the language has dependent classes, the *CT* function takes an environment Γ that maps variables to types. Γ is a finite *ordered* list of $x:T$ pairs in the order in which they came into scope. To be well-formed, an environment Γ may contain at most one pair $x:T$ for a given x.

In addition to returning a class declaration for a type, *CT* also interprets the `extends` clause of the class declaration, replacing any occurrences of `This` with the actual enclosing class. This type is passed as the second argument to *CT*. Thus, $CT(\Gamma, T_0, T)$ returns the interpreted class declaration for T in an environment Γ where T_0 is substituted into the `extends` clause of member classes of the class declaration. To save space, we write C `ext` T $\{\overline{L}\ \vec{F}\ \overline{M}\}$ to represent `class` C `extends` T $\{\overline{L}\ \vec{F}\ \overline{M}\}$.

Classes inherit member classes of the base class into the body of the derived class. The set $\overline{L}_1 \bullet \overline{L}_2$, defined in Figure 10, merges the class bodies of identically named classes in $\overline{L}_1$ and $\overline{L}_2$, creating class declarations for implicit classes when needed. Classes in $\overline{L}_1$—classes inherited from the base class—are overridden by classes in $\overline{L}_2$—nested classes of the derived class. Fields and methods of classes defined in a base class are *not* copied when the nested class is inherited into the subclass; they can be found by the member lookup functions defined in Figure 11.

The function $classes(\Gamma, T_0, T)$ defined in Figure 11 returns the set of member classes of T with T_0 substituted for `This` in the `extends` clause of the member classes.

The rules CT-OUTER and CT-NEST define the *CT* function for top-level classes C and nested classes $T.C$, respectively, using the top-level class table *TCT*. The three rules CT-RUNTIME, CT-PRE, and CT-DEP return class declarations for dependent classes and prefix types. In these rules, the *CT* function returns for type T an *anonymous class declaration* whose superclass is a simple class type P bounding T.[3] Member classes are copied down into the anonymous class declaration as with top-level and nested classes.

In each rule, the type T_0 is substituted for `This` in the `extends` clauses of nested classes. For $L = C$ `ext` S $\{\overline{L}\ \vec{F}\ \overline{M}\}$, we define $L\{T_0/\texttt{This}\}$ as C `ext` $S\{T_0/\texttt{This}\}$ $\{\overline{L}\ \vec{F}\ \overline{M}\}$, and we define $S\{T_0/\texttt{This}\}$ as:

$$
\begin{aligned}
C\{T_0/\texttt{This}\} &= C \\
S.C\{T_0/\texttt{This}\} &= S\{T_0/\texttt{This}\}.C \\
\texttt{This}\{T_0/\texttt{This}\} &= T_0 \\
P\,[S:P.C]\,\{T_0/\texttt{This}\} &= \mathit{prefix}(P, P', P', P.C) \\
&\qquad \text{where } S\{T_0/\texttt{This}\} = P' \\
P\,[S:P.C]\,\{T_0/\texttt{This}\} &= P\,[T:P.C] \\
&\qquad \text{where } S\{T_0/\texttt{This}\} = T \neq P' \\
&\qquad \text{for any } P'
\end{aligned}
$$

The function *prefix* is defined in Figure 10 and is used to ensure the type produced by the substitution is well-formed.

The rule CT-RUNTIME defines class lookup for types whose exact run-time class can be determined statically. The partial function *exact-class*, defined in Figure 10, returns a simple class type P for these types. *exact-class* is only defined only for dependent classes and prefix types containing access paths of the form $\ell_P.\texttt{class}$. Since these types are not valid surface syntax CT-RUNTIME is not used when type-checking the program, but is needed to prove the type system sound.

The rule CT-PRE defines class lookup for prefix types $P\,[T:P.C]$ whose run-time class is *not* statically known. An anonymous class declaration whose superclass is P is returned.

Similarly, the rule CT-DEP defines class lookup for dependent classes $p.\texttt{class}$ whose run-time class is *not* statically known by returning an anonymous class declaration whose superclass is the declared type of p.

The judgment $\Gamma \vdash p$ `final` T, defined in Figure 10, is used to check that an access path has type T and is immutable. The

[3] Anonymous class declarations should not be confused with Java anonymous classes.

$$\frac{\mathit{super}(\Gamma, T) = T'}{\Gamma \vdash T \leq T'} \quad (\leq\text{-EXTENDS})$$

$$\frac{\Gamma \vdash T \leq T'}{\Gamma \vdash T.C \leq T'.C} \quad (\leq\text{-NEST})$$

$$\frac{\mathit{exact\text{-}class}(T) = P \quad \mathit{exact\text{-}class}(T') = P}{\Gamma \vdash T \leq T'} \quad (\leq\text{-RUNTIME})$$

Figure 12: Subtyping

rules for $\Gamma \vdash p$ `final` T and for $CT(\Gamma, T_0, T)$ are mutually recursive (via the definition *ftype*, defined in Figure 11). For a dependent class p.`class` to be well-formed, the static type of p must be a simple type P; this restriction is sufficient to ensure the definition of CT for dependent classes is well-founded. As in [27], we wish to ensure that no type information is lost when typing a final access path so that we can tightly bound p.`class`. Consequently, there is no subsumption rule that can be used to prove $\Gamma \vdash p$ `final` T. Rules F-LOC and F-VAR bound the types of locations and local variables, respectively. F-LOC requires that the type of the location ℓ_P be well-formed according to the rules in Figure 13. Rule F-NULL states that the `null` value may have any type. Rule F-GET uses the *ftype* function to retrieve the type of the field. The target of a field access in a final access path must be `final`. Finally, the rule F-RUNTIME permits two types with the same run-time class (if statically known) to be considered to have the same type.

6.3 Method and field lookup

Method and field lookup functions are defined in Figure 11. The functions are defined using the linearization of superclasses described informally in Section 3. The ordering, $\mathit{ord}(\Gamma, T)$, is defined so that classes that T overrides occur before T's declared superclass, $\mathit{super}(\Gamma, T)$. The function is used to iterate through the superclasses to locate the most-specific method or field definition.

In Figure 11, the function $\mathit{fields}(\Gamma, T_0, T)$ returns all fields declared in class T_0 or superclasses of T_0, iterating through superclasses of T_0 using the function, beginning with T. Auxiliary functions *ftype*, *finit*, and *fnames* are defined from *fields* to return the type of a given field, the initializer of a field, and the set of all field names for a given class, respectively. The function $\mathit{method}(\Gamma, T_0, T, m)$ returns the most-specific method declaration for method m, iterating through the superclasses of T_0, beginning with T. Functions *mbody* and *mtype* return the method body and method type, respectively, for a method.

6.4 Operational semantics

The operational semantics of the language are given in Figure 11. The semantics are defined using a reduction relation $\longrightarrow$ that maps a configuration of a heap H and expression e to a new configuration. A heap H is a function from memory locations ℓ_P to objects $P\ \{\overline{f = \ell_{P'}}\}$. The notation $\langle H, e\rangle \longrightarrow \langle H', e'\rangle$ means that expression e and heap H step to expression e' and heap H'. The initial configuration for program $\langle TCT, e\rangle$ is $\langle \emptyset, e\rangle$. Final configurations are of the form $\langle H, \ell_P\rangle$ or $\langle H, \texttt{null}\rangle$.

The reduction rules are mostly straightforward. R-CALL and R-SUPER use the *mbody* function defined in Figure 10 to locate the most specific implementation of m. Recall that `super` calls are annotated with the name of lexically enclosing class containing the call. R-SUPER uses the function, defined in Figure 11 to start the search for the method body at the next-most specific method after the lexically enclosing class Q.

For a `new` T `as` x expression, R-NEW allocates an object of the run-time class P of type T. The rule initializes all fields of the new object to `null` and then steps to a sequence of field assignments to initialize the expression, and finally evaluates to the location of the newly allocated object. The field assignments are annotated with the keyword `final` to indicate that it is permitted to assign to `final` fields. Since final assignments are not permitted in the surface syntax, `final` fields may only be assigned once. The field initializers $\vec{e}$ appearing explicitly in the `new` expression are evaluated with the new location substituted for x. The other fields of the object are initialized using the default initializers $\vec{e'}$ with the new location substituted for `this`.

The run-time class of T is computed using the function *runtime-class*, defined in Figure 10. For prefix types P`[`T'`:`$P.C$`]`, *runtime-class* uses the *prefix* function to compute the run-time class of the prefix type by iterating through the superclasses of T' until a class overriding $P.C$ is found; the container of this class is the run-time class of the prefix type.

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, simulating a Java `NullPointerException`.

6.5 Static semantics

The static semantics of the language are defined by rules for subtyping, type well-formedness, typing, and conformance.

Subtyping

The subtyping relation is the smallest reflexive, transitive relation consistent with the rules in Figure 12. Rule $\leq$-EXTENDS says that a class is a subtype of its declared superclass. The subtyping relationships for dependent classes and prefix types are covered by $\leq$-EXTENDS. Rule $\leq$-NEST says that a nested class C in T is a subclass of the class C in T' that it overrides. Finally, rule $\leq$-RUNTIME states that two types are subtypes of each other if their run-time classes are equal.

Type well-formedness

Since types may depend on variables, we define type well-formedness in Figure 13 with respect to an environment Γ, written $\Gamma \vdash T$ wf. A non-dependent type is well-formed if a class declaration for it can be located through the *TCT*. A dependent class p.`class` is well-formed if p is `final` and has a simple non-dependent class type P. A prefix type P`[`T`:`$P.C$`]` is well-formed if its subterms are well-formed and if T is an *exact type* and is also a subtype of $P.C$. The last requirement ensures the run-time class of the type can be determined.

A type is *exact* if it is a dependent class or a prefix type. The subtyping rules ensure that no type can be proved a subtype of an exact type. This restriction ensures that a variable of type p.`class` can be assigned only values with the same run-time class as the object referred to by p. The restriction does not limit expressiveness since non-exact prefix types can be desugared to either exact prefix types or to non-prefix types.

$$\frac{\mathit{runtime\text{-}class}(T) = P \quad \vdash T \text{ wf} \quad \vdash P \text{ wf}}{\vdash \ell_P : T} \quad \text{(T-LOC)}$$

$$\frac{\Gamma \vdash T \text{ wf}}{\Gamma \vdash \texttt{null} : T} \quad \text{(T-NULL)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \text{(T-VAR)}$$

$$\frac{\Gamma \vdash e : T \quad \Gamma, x : T \vdash e' : T' \quad \Gamma \vdash T \text{ wf} \quad \Gamma \vdash T' \text{ wf} \quad x \notin \mathrm{dom}(\Gamma)}{\Gamma \vdash \texttt{final } T\ x = e;\ e' : T'} \quad \text{(T-LET)}$$

$$\frac{\Gamma \vdash p\ \texttt{final}\ T \quad \mathit{ftype}(\Gamma, T, f_i) = [\texttt{final}]\ T_i}{\Gamma \vdash p.f_i : T_i\{p/\texttt{this}\}} \quad \text{(T-GET)}$$

$$\frac{\Gamma \vdash p\ \texttt{final}\ T \quad \Gamma \vdash e : T_i\{p/\texttt{this}\} \quad \mathit{ftype}(\Gamma, T, f_i) = [\texttt{final}]\ T_i \quad \Gamma \vdash e' : T'}{\Gamma \vdash p.f_i =_{[\texttt{final}]} e;\ e' : T'} \quad \text{(T-SET)}$$

$$\frac{\Gamma \vdash p\ \texttt{final}\ T \quad \mathit{mtype}(\Gamma, T, T, m) = (\vec{x} : \vec{T}) \to T' \quad \Gamma \vdash \vec{v} : \vec{T}\{p/\texttt{this}, \vec{v}/\vec{x}\}}{\Gamma \vdash p.m(\vec{v}) : T'\{p/\texttt{this}, \vec{v}/\vec{x}\}} \quad \text{(T-CALL)}$$

$$\frac{\Gamma \vdash P \text{ wf} \quad \Gamma \vdash v_0 : P \quad \mathit{mtype}(\Gamma, P, \mathit{super}(P), m) = (\vec{x} : \vec{T}) \to T' \quad \Gamma \vdash \vec{v} : \vec{T}\{v_0/\texttt{this}, \vec{v}/\vec{x}\}}{\Gamma \vdash v_0.\texttt{super}_P.m(\vec{v}) : T'\{v_0/\texttt{this}, \vec{v}/\vec{x}\}} \quad \text{(T-SUPER)}$$

$$\frac{\mathit{ftype}(\Gamma, T, \overline{f}) = \overline{T} \quad \Gamma, x : T \vdash \overline{e} : \overline{T}\{x/\texttt{this}\}}{\Gamma \vdash \texttt{new } T \texttt{ as } x\ \{\overline{f} = \overline{e}\} : T} \quad \text{(T-NEW)}$$

$$\frac{\Gamma \vdash p\ \texttt{final}\ P}{\Gamma \vdash p : p.\texttt{class}} \quad \text{(T-DEP)}$$

$$\frac{\Gamma \vdash e : T \quad \Gamma \vdash T \leq T'}{\Gamma \vdash e : T'} \quad \text{(T-}\leq\text{)}$$

Figure 14: Static semantics

$$\frac{C \in \mathrm{dom}(\mathit{TCT})}{\Gamma \vdash C \text{ wf}} \quad \text{(WF-OUTER)}$$

$$\frac{\Gamma \vdash T \text{ wf} \quad \mathit{classes}(\Gamma, T, T) = \overline{L_s} \quad C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\} \in \overline{L_s}}{\Gamma \vdash T.C \text{ wf}} \quad \text{(WF-NEST)}$$

$$\frac{\Gamma \vdash p\ \texttt{final}\ P}{\Gamma \vdash p.\texttt{class wf}} \quad \text{(WF-DEP)}$$

$$\frac{\Gamma \vdash P.C \text{ wf} \quad \Gamma \vdash T \text{ wf} \quad \mathit{is\text{-}exact}(T) \quad \Gamma \vdash T \leq P.C}{\Gamma \vdash P\,[T : P.C] \text{ wf}} \quad \text{(WF-PRE)}$$

$$\mathit{is\text{-}exact}(T) = \begin{cases} \mathit{false} & \text{if } T = C \vee T = T'.C \\ \mathit{true} & \text{otherwise} \end{cases}$$

Figure 13: Type well-formedness

Typing

The typing rules are shown in Figure 14. The typing context consists of an environment Γ. The typing judgment $\Gamma \vdash e : T$ is used to type-check expressions.

Rules T-NULL and T-VAR are standard. The rule T-LOC allows a location of type P to be used as a member of any type T where $\mathit{runtime\text{-}class}(T) = P$. This rule helps to ensure types are preserved across the evaluation of a `new` expression.

The rule T-LET type-checks a local variable initialization expression. The declared type T must be well-formed in the environment Γ. The expression e' following the declaration is type-checked with the new variable in scope. The type of e' must be well-formed in the *original* environment to ensure that its type does not depend on the new variable, which is not in scope outside of e'.

Rules T-GET and T-SET use the *ftype* function to retrieve the type of the field. The target of a field access or assignment must be a `final` path, permitting substitution to be performed on the field type: occurrences of `this` in the field type are replaced with the actual target p. Rule T-SET permits assignment to `final` fields, but only for assignments annotated with `final`. This enables `final` fields to be initialized, but not assigned to arbitrarily.

Rules T-CALL and T-SUPER are used to check calls. The function *mtype* returns the method's type. The method type may depend on `this` or on its parameters $\vec{x}$, which are considered part of the method type. The receiver must be `final` to permit substitution for argument and return types dependent on `this`. The arguments are also substituted into the type.

Rule T-NEW is used to check a `new` expression. The fields used in the inline constructor body must be declared in the class being allocated and the initializers must have the appropriate types. Since the initializers use x to refer to the newly allocated object, x is substituted for `this` in the field types.

Rule T-DEP allows any final access path with a simple nested class type to take on a dependent type. Finally, rule T-$\leq$ is the usual subsumption rule for subtyping.

Declarations

To initiate type-checking, declarations are checked as shown in Figure 15. The program is checked with rule OK-PROGRAM, which checks every class in the *TCT* and type-checks the "main" expression e in an empty environment.

Rule OK-CLASS type-checks a class declaration of the form C `ext` $S\ \{\overline{L}\ \vec{F}\ \overline{M}\}$, nested within a class P, where P is possibly ε (i.e., C is top-level). Type-checking recurses on all member declarations including nested classes. The rule also checks member classes and methods for conformance with the corresponding declarations in their superclass. To ensure no other type can be proved a subtype of a dependent class or of a prefix type, it is required that a class cannot be declared to extend the type schema `This` or any prefix of `This`. This requirement is enforced by substituting `this.class` for the

$$\frac{\vdash \overline{L} \text{ ok in } \varepsilon \quad \vdash e : T}{\vdash \langle \overline{L}, e \rangle \text{ ok}} \qquad \text{(OK-PROGRAM)}$$

$$\frac{\begin{array}{c} \vdash \overline{L} \text{ ok in } P.C \\ \vdash \vec{F} \text{ ok in } P.C \\ \vdash \overline{M} \text{ ok in } P.C \\ \mathit{classes}(\emptyset, S\{P/\texttt{This}\}, S\{P/\texttt{This}\}) = \overline{L_s} \\ \left(\begin{array}{c} C \in \mathrm{dom}(\overline{L}) \wedge C \in \mathrm{dom}(\overline{L_s}) \\ \Rightarrow \vdash L(C) \text{ in } P.C \text{ overrides class of } S\{P/\texttt{This}\} \end{array}\right) \\ \vdash \overline{M} \text{ in } P.C \text{ overrides method of } S\{P/\texttt{This}\} \\ \texttt{this}:P \vdash S\{\texttt{this.class}/\texttt{This}\} \text{ wf} \\ \neg \mathit{is\text{-}exact}(S\{\texttt{this.class}/\texttt{This}\}) \end{array}}{\vdash C \texttt{ ext } S \ \{\overline{L}\ \vec{F}\ \overline{M}\} \text{ ok in } P} \qquad \text{(OK-CLASS)}$$

$$\frac{\begin{array}{c} \mathit{super}(\{\texttt{this} : P_s\}, \texttt{this.class}.C) = T_s \\ \mathit{classes}(\emptyset, S\{P/\texttt{This}\}, S\{P/\texttt{This}\}) = \overline{L_s} \\ \left(\begin{array}{c} C \in \mathrm{dom}(\overline{L}) \wedge C \in \mathrm{dom}(\overline{L_s}) \\ \Rightarrow \vdash L(C) \text{ in } P.C \text{ overrides class of } S\{P/\texttt{This}\} \end{array}\right) \\ \vdash \overline{M} \text{ in } P.C \text{ overrides method of } P_s.C \\ \texttt{this}:P \vdash S\{\texttt{this.class}/\texttt{This}\} \leq T_s \end{array}}{\vdash C \texttt{ ext } S \ \{\overline{L}\ \vec{F}\ \overline{M}\} \text{ in } P \text{ overrides class of } P_s} \qquad \text{(OV-CLASS)}$$

$$\frac{\texttt{this}:P \vdash T \text{ wf} \quad \texttt{this}:P \vdash e : T}{\vdash [\texttt{final}]\ T\ f = e \text{ ok in } P} \qquad \text{(OK-FIELD)}$$

$$\frac{\begin{array}{c} \texttt{this}:P, x_1 : T_1, \ldots, x_{i-1} : T_{i-1} \vdash T_i \text{ wf} \\ \texttt{this}:P, \vec{x}:\vec{T} \vdash T_0 \text{ wf} \\ \texttt{this}:P, \vec{x}:\vec{T} \vdash e : T_0 \end{array}}{\vdash T_0\ m(\vec{T}\ \vec{x})\ \{e\} \text{ ok in } P} \qquad \text{(OK-METHOD)}$$

$$\frac{\begin{array}{c} \mathit{mtype}(\emptyset, P, P_s, m) = (\vec{x'} : \vec{T'}) \rightarrow T_0' \\ \Rightarrow \vec{T'} = \vec{T}\{\vec{x'}/\vec{x}\} \wedge T_0' = T_0\{\vec{x'}/\vec{x}\} \end{array}}{P \vdash T_0\ m(\vec{T}\ \vec{x})\ \{e\} \text{ overrides method of } P_s} \qquad \text{(OV-METHOD)}$$

Figure 15: Checking declarations

schema `This` in the superclass S; and checking that this type is well-formed and not an exact type.

Rule OV-CLASS checks that a class declaration conforms to any class declarations it overrides. When overriding a class with superclass T_s, it is required that the new superclass $S\{\texttt{this.class}/\texttt{This}\}$ be a subtype of T_s in the typing environment $\texttt{this}:P$. This restriction differentiates nested class overriding from arbitrary multiple inheritance.

Rule OK-FIELD states that in the body of class P, a field declaration of the form $[\texttt{final}]\ T\ f = e$ type-checks if the type T is well-formed and the initializer e type-checks in an environment where `this` has type P. For simplicity, we assume a field named f is declared at most once in the program, and we assume all methods and nested classes are uniquely named up to overriding.

Rule OK-METHOD checks that each parameter type T_i is well-formed in an environment that includes only `this` and the parameters to the left of T_i. The method body must have the same type as the declared return type. As in Java, method types are invariant; OV-METHOD enforces this requirement.

6.6 Soundness

Our soundness proof is structurally similar to the proof of soundness for Featherweight Java (FJ) [15]. The proof uses the standard technique of proving subject reduction and progress lemmas [35]. The key lemmas are stated here. The complete proof is in the appendix.

Subject reduction

Because expressions in our language are evaluated in a heap, to state the subject reduction lemma, we first define a well-typedness condition for heaps and for configurations $\langle H, e \rangle$.

Definition 6.1 (Well-typed heaps) A heap H is *well-typed* if for any memory location $\ell_P \in \mathrm{dom}(H)$,

- $H(\ell_P) = P\ \{\overline{f} = \overline{\ell_{P'}}\}$,
- $\vdash \mathit{ftype}(\emptyset, P, \overline{f}) = \overline{T}$,
- $\vdash \overline{\ell_{P'}} : \overline{T}\{\ell_P/\texttt{this}\}$, and
- $\overline{\ell_{P'}} \subseteq \mathrm{dom}(H)$

Definition 6.2 (Well-formed configurations) A configuration $\langle H, e \rangle$ is *well-formed* if H is well-typed and for any location ℓ_P free in e, $\ell_P \in \mathrm{dom}(H)$.

The subject reduction lemma states that a step taken in the evaluation of a well-formed configuration results in a well-formed configuration.

Lemma 6.3 (Subject reduction) Suppose $\vdash e : T$, $\langle H, e \rangle$ is well-formed, and $\langle H, e \rangle \longrightarrow \langle H', e' \rangle$. Then $\vdash e' : T$ and $\langle H', e' \rangle$ is well-formed.

Progress

The progress lemma states that for any well-formed configuration $\langle H, e \rangle$, either e is a base value ℓ_P or `null`, or $\langle H, e \rangle$ can make a step according to the operational semantics.

Lemma 6.4 (Progress) If $\vdash e : T$, $\vdash T$ wf, $\langle H, e \rangle$ is well-formed, then either $e = b$ or there is a configuration $\langle H', e' \rangle$ such that $\langle H, e \rangle \longrightarrow \langle H', e' \rangle$.

Soundness

Finally, we define the normal form of a configuration, define well-formedness for programs, and state the soundness theorem.

Definition 6.5 (Normal forms) A configuration $\langle H, e \rangle$ is in *normal form* if there is no $\langle H', e' \rangle$ such that $\langle H, e \rangle \longrightarrow \langle H', e' \rangle$.

Definition 6.6 A program $Pr = \langle TCT, e \rangle$ is *well-formed* if $\vdash TCT$ ok and $\emptyset \vdash e : T$ for some T such that $\emptyset \vdash T$ wf.

Theorem 6.7 (Soundness) Given a well-formed program $Pr = \langle TCT, e \rangle$, if the configuration $\langle \emptyset, e \rangle$ is well-formed and $\vdash e : T$, and if $\langle H', e' \rangle$ is a normal form such that $\langle \emptyset, e \rangle \longrightarrow^* \langle H', e' \rangle$, then e' is either a location $\ell_P \in \mathrm{dom}(H')$ or `null` and $\vdash e' : T$.

7 Related work

Over the past decade a number of mechanisms have been proposed to provide object-oriented languages with additional extensibility. Nested inheritance uses ideas from many of these other mechanisms to create a flexible and largely transparent mechanism for code reuse.

Virtual classes

Nested inheritance is related to virtual types and virtual classes. Virtual types were originally developed for the language Beta [20, 21], primarily as a mechanism for generic programming rather than for extensibility. Later work proposed virtual types as a means of providing genericity in Java [33].

Nested classes in Jx are similar, but not identical, to virtual classes. Unlike virtual classes, nested classes in Jx are attributes of their enclosing class, not attributes of *instances* of their enclosing class. Suppose class `A` has a nested class `B` and that `a1` and `a2` are references to instances of possibly distinct subclasses of `A`. The virtual classes `a1.B` and `a2.B` are distinct classes. In contrast, the Jx types `a1.class.B` and `a2.class.B` may be considered equivalent if it can be proved, either statically or at run-time, that `a1` and `a2` refer to instances of the same class.

Virtual types are not statically safe because they permit method parameter types to change covariantly with subtyping, rather than contravariantly. Beta and other languages with virtual types insert run-time checks when a method invocation cannot be statically proved sound. Dependent classes in Jx provide the expressive power of covariant method parameter types without introducing unsoundness. Recent work on type-safe variants of virtual types has limited method parameter types to be invariant [34] and used *self types* [4] as discussed below.

Nested inheritance supports a form of virtual superclasses; nested classes may extend other nested classes referred to by `This`, providing mixin-like functionality. The language Beta does not support virtual superclasses, but gbeta [8] does.

As discussed in Section 3, nested inheritance does not support generic types. A nested class may only be declared a subtype of another type (via the class's `extends` clause), not *equal* to another type. Generic types may be used to provide genericity, which is already supported in Java through parameterized types. To ensure inheritance relationships can be determined statically, a virtual type in Beta may be inherited from only if it is *final bound*. Since nested classes in Jx are `static`, Jx does not permit inheritance from dependent classes, ensuring a static inheritance hierarchy.

Igarashi and Pierce [14] model the semantics of virtual types and several variants in a typed lambda-calculus with subtyping and dependent types.

The work most closely related to nested inheritance is Odersky et al.'s language Scala [26, 37], which supports scalable extensibility through a statically safe virtual type mechanism and path-dependent types similar to Jx's dependent classes. However, Scala's path dependent type *p*`.type` is a singleton type containing only the value named by access path *p*; our *p*`.class` is not a singleton: `new x.class`(...), for instance, creates a new object of type `x.class` distinct from the object referred to by *p*. This difference gives Jx more flexibility, while preserving type soundness. Scala has no analogue to prefix types.

Scala permits extensions to be composed through mixins. Jx supports mixin-like functionality via virtual superclasses. With nested inheritance, several mixins can be applied at once to a collection of nested classes by overriding the base class (or base package) of their container. In contrast, Scala requires the programmer to explicitly name the superclass of each individual mixin when it is applied.

Family polymorphism

Ernst [9] introduces the term *family polymorphism* to describe polymorphism that allows reuse of groups of mutually dependent classes, that is a *family* of classes. The basic idea is to use an object as a repository for a family of classes. Virtual classes of the same object are considered part of the same family. The language gbeta [8], as well as Scala [26], described above, provides family polymorphism using a dependent type system that prevents the confusion of classes from different families. Nested inheritance is a limited form of family polymorphism. In the original formulation, each *object* defines a distinct family consisting of its nested classes. With nested inheritance, since nested classes are associated with an enclosing class rather than with an instance of the enclosing class, each *class* defines a distinct family. Thus, nested inheritance permits only a finite number of families. However, consider the case of a class `A` with nested class `B` and references `a1` and `a2` of type `A`. If `a1.class` and `a2.class` cannot be shown statically to have the same type, then `a1.class.B` and `a2.class.B` may be considered to be of distinct families, although at run-time they may be of the same family. Jx allows objects to be passed between the two families by casting `a1.class` to `a2.class` or vice versa. This added flexibility enables greater reuse. Moreover, using prefix types, a family need not be identified solely be a single object. In gbeta, an explicit representative of the family must be passed around. It lacks an analogy to prefix types, which enable a member of a family to unambiguously identify that family.

Delegation layers [29] use virtual classes and delegation to provide family polymorphism, solving many of the problems of mixin layers. With normal inheritance and virtual classes, when a method is not implemented by a class, the call is dispatched to the superclass. With delegation, the superclass view of an object may be implemented by another *object*. Methods are dispatched through a chain of delegate objects rather than through the class hierarchy. Delegation layers provide much of the same power as nested inheritance. Since delegates are associated with objects at run-time rather than at compile-time, delegation allows objects to be composed more flexibly than with mixins or with nested inheritance. However, no formal semantics has been given for delegation layers, and because delegation layers rely on virtual classes, they are not statically type-safe.

Higher-order hierarchies

Nested inheritance is similar to Ernst's higher-order hierarchies [10]. Like nested inheritance, higher-order hierarchies support family polymorphism. Additionally, when a subclass `A2` overrides a nested class `B` of `A2`'s base class `A`, the overriding class `A2.B` inherits from `A.B`. However, unlike nested inheritance, there is no subtyping relationship between `A.B` and `A2.B`. By ensuring `A2.B` is a subtype of `A.B`, nested inheritance permits more code reuse. Like nested inheritance, the inheritance hierarchy can be modified by overriding the superclass of a nested class.

Other nested types

Nested classes originated with Simula [7].

Igarashi and Pierce [16] present a formalization of Java's inner classes, using Featherweight Java [15]. An instance of a Java inner class holds a reference to its enclosing instance. If inner classes are permitted in Jx, a translation similar to Igarashi and Pierce's can be applied, where if inner class `C` has an immediately enclosing instance of class `P`, then the translation of `C` has a final field of type `P[this.class]`.

Odersky and Zenger [28] propose nested types, which combine the abstraction properties of ML-style modules with support, through encoding, for object-oriented constructs like virtual types, self types, and covariant families of classes.

Self types and matching

Bruce et al. [5, 3] introduce *matching* as an alternative to subtyping in an object oriented language. With matching, the *self type*, or `MyType`, can be used in a method signature to represent the run-time class of the method's receiver. To permit `MyType` to be used for method parameters, type systems with `MyType` decouple subtyping and subclassing. In PolyTOIL and LOOM, a subclass *matches* its base class but is not a subtype. Although there is no explicit notion of matching in our type system, the rules for subtyping and type equivalence given here have a similar effect. The *p*`.class` construct provides similar functionality to `MyType`, but is more flexible since it permits `this.class` to escape the body of its class by assigning `this.class` into another variable or returning a value of that type from a method.

Mixins

A *mixin* [2, 11], also known as an *abstract subclass*, is a class parameterized on its superclass. Mixins are able to provide uniform extensions, such as adding new fields or methods, to a large number classes. Recent work has extended Java with mixin functionality [22, 1]. Because nested inheritance as described here has no type parametricity, it cannot provide a mixin that can be applied to many different, unrelated classes, Nested inheritance does, however, provides mixin-like functionality by allowing the superclass of an existing base class to be changed or fields and methods to be added by overriding the class's superclass through extension of the superclass's container. Additionally, nested inheritance allows the implicit subclasses of the new base class to be instantiated without writing any additional code. Mixins have no analogous mechanism.

Mixin layers [31] are a generalization of mixins to multiple classes. A mixin layer is a design pattern for implementing a group of interrelated mixin classes and extending them while preserving their dependencies. Mixin layers do not provide family polymorphism. Delegation layers [29], described above, were designed to overcome this limitation through a new language mechanism.

Open classes

An *open class* [6] 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 is also able to add new methods to a class without the need for recompilation of clients of the class, provided that the class is nested in a container that can be extended, and that clients of the class refer to it using dependent types. Nested inheritance provides additional extensibility that open classes do not, such as the "virtual" behavior of constructors. An important difference is that open classes *modify* existing class hierarchies. The original hierarchy and the modified hierarchy cannot coexist within the same program. Nested inheritance creates a new class hierarchy by extending the container of the classes in the hierarchy, permitting use of the original hierarchy in conjunction with the new one.

Aspect-oriented programming

Aspect-oriented programming (AOP) [18, 17] is concerned with the management of *aspects*, functionality that crosscuts standard modular boundaries. Nested inheritance provides aspect-like extensibility, in that an extension to a container may implement functionality that cuts across the class boundaries of the nested classes. Like open classes, aspects modify existing class hierarchies, preventing the new hierarchy from being used alongside the old.

8 Conclusions

Nested inheritance is an expressive yet unobtrusive mechanism for writing highly extensible frameworks. It provides the ability to inherit a collection of related classes while preserving the relationships among those classes, and it does so without sacrificing type safety or imposing new run-time checks. The use of dependent classes and prefix types enables reusable code to unambiguously yet flexibly refer to components on which it depends. Nested inheritance is fundamentally an inheritance mechanism rather than a parameterization mechanism, which means that every name introduced by a component becomes a possible implicit hook for future extension. Therefore extensible code does not need to be burdened by explicit parameters that attempt to capture all the ways in which it might be extended later.

We formalized the essential aspects of nested inheritance in an object calculus with an operational semantics and type system, and were able to show that this type system is sound. Thus extensibility is obtained without sacrificing compile-time type safety.

Our experience with implementing extensible frameworks gives us confidence that nested inheritance will prove useful. We defined a language Jx that incorporates the nested inheritance mechanism and implemented a prototype compiler for the core mechanisms of this language. The translation implemented by this compiler does not duplicate inherited code. The next step is clearly to complete the Jx implementation; we look forward to using it to build the next version of Polyglot.

Acknowledgments

Michael Clarkson and Jed Liu participated in early design discussions. Matthew Fluet, Michael Clarkson, Jens Palsberg, and the anonymous reviewers provided thorough and insightful comments.

This research was supported in part by ONR Grant N00014-01-1-0968, NSF Grants 0208642 and 0133302, and an Alfred P. Sloan Research Fellowship. Nathaniel Nystrom was supported by an Intel Foundation Ph.D. 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] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of Java with mixins. In *Proc. ECOOP '00*, LNCS 1850, pages 154–178, Cannes, France, 2000.

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

[3] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *Proceedings of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, number 1241 in Lecture Notes in Computer Science, pages 104–127, Jyväskylä, Finland, June 1997. Springer-Verlag.

[4] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming (ECOOP)*, number 1445 in Lecture Notes in Computer Science, pages 523–549, Brussels, Belgium, July 1998. Springer-Verlag.

[5] 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.

[6] 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.

[7] O.-J. Dahl et al. The Simula 67 common base language. Publication No. S-22, Norwegian Computing Center, Oslo, 1970.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 2nd edition, 2000. ISBN 0-201-31008-2.

[14] 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.

[15] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[16] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. *Information and Computation*, 177(1):56–89, August 2002.

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[18] 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.

[19] B. Liskov et al. CLU reference manual. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*, volume 114. Springer-Verlag, Berlin, 1981.

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

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

[22] Sean McDirmid, Matthew Flatt, and Wilson C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proc. OOPSLA '01*, October 2001.

[23] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[24] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif: Java information flow. Software release. Located at `http://www.cs.cornell.edu/jif`, July 2001–2003.

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

[26] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language, June 2004. `http://scala.epfl.ch/-docu/files/ScalaOverview.pdf`.

[27] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of 17th European Conference on Object-Oriented Programming (ECOOP 2003)*, number 2743 in Lecture Notes in Computer Science, pages 201–224. Springer-Verlag, July 2003.

[28] Martin Odersky and Christoph Zenger. Nested types. In *8th Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2001.

[29] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110, Málaga, Spain, 2002. Springer-Verlag.

[30] 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 [**?**], pages 13–23.

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

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

[33] Kresten Krab Thorup. Genericity in Java with virtual types. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, number 1241 in Lecture Notes in Computer Science, pages 444–471. Springer-Verlag, 1997.

[34] Mads Torgerson. Virtual types are statically safe. In *5th Workshop on Foundations of Object-Oriented Languages (FOOL)*, January 1998.

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

[36] Matthias Zenger and Martin Odersky. Extensible algebraic datatypes with defaults. In *Proc. 6th ACM SIGPLAN International Conference on Functional Programming (ICFP)*, Firenze, Italy, September 2001.

[37] Matthias Zenger and Martin Odersky. Independently extensible solutions to the expression problem. Technical Report IC/2004/33, École Polytechnique Fédérale de Lausanne, March 2004.

A Soundness proof

Our soundness proof is structurally similar to the proof of soundness for Featherweight Java (FJ) [15]. The proof uses the standard technique of proving subject reduction and progress lemmas [35]. The key lemmas are stated here. The complete proof is in the appendix.

A.1 Canonical derivations

Several of the inference rules admit infinite derivations. For instance, it is possible to have an infinite sequence of instances of the subsumption rule T-$\leq$, using the reflexivity of subtyping. To ensure that induction on derivations is well-founded, we define *canonical derivations* for each of the typing judgments by restricting derivations in following ways:

- Reflexivity of $\leq$ is not used in the derivation.
- No use of T-$\leq$ can have as a premise a derivation ending in T-$\leq$.
- No use of F-RUNTIME with conclusion $\Gamma \vdash p$ `final` T can have the same judgment as its premise.
- No use of F-RUNTIME can have as a premise a derivation ending in F-RUNTIME.

Induction on canonical derivations is well-founded. For the remainder of the proof, we assume derivations are canonical.

By restricting the type-checking algorithm to canonical derivations, the type-checking is decidable.

A.2 Conformance

To prove the subject reduction lemma, we need to ensure that the signature information retrieved from the class table CT was type-checked. We first define the natural extension of TCT to all non-dependent types P.

Definition ??

$$TCT^*(C) = TCT(C)$$

$$\frac{TCT^*(P) = C_t \texttt{ ext } S_t \ \{\overline{L_t}\ \vec{F_t}\ \overline{M_t}\} \qquad C \texttt{ ext } S\ \{\overline{L}\ \vec{F}\ \overline{M}\} \in \overline{L_t}}{TCT^*(P.C) = C \texttt{ ext } S\ \{\overline{L}\ \vec{F}\ \overline{M}\}}$$

Lemma A.1 (Nested classes checked) If $P = Q.C$ with Q possibly ε and $P \in \mathrm{dom}(TCT^*)$, then $\vdash TCT^*(P)$ ok in Q.

Proof. The proof is simple, by induction on the structure of P. □

The next two lemmas state that a method body or field initializer has the proper type. These lemmas follow almost immediately from Lemma A.1.

Lemma A.2 (Method conformance) If $mtype(\emptyset, P, Q, m) = (\vec{x} : \vec{T}) \to T'$, and $mbody(P, Q, m) = (\vec{x}, e)$, then there is a P' not before Q in $ord(\emptyset, P)$ where $\vdash P \leq P'$ such that $\texttt{this} : P', \vec{x} : \vec{T} \vdash e : T'$.

Proof. The proof is by induction on the derivation of $mbody(P, Q, m)$. In the base case, assume $CT(\emptyset, Q, Q) = C \texttt{ ext } T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}$ and $T'\ m(\vec{T}\ \vec{x})\ \{e\} \in \overline{M}$. Since $\overline{M}$ is not empty, $Q \in \mathrm{dom}(TCT^*)$. Thus by Lemma A.1, if $Q = Q'.C$ (Q' possibly ε), then $\vdash TCT^*(Q)$ ok in Q'. It follows immediately that $\vdash T'\ m(\vec{T}\ \vec{x})\ \{e\}$ ok in Q, and thus $\texttt{this} : Q, \vec{x} : \vec{T} \vdash e : T'$. To complete the base case, observe that $\vdash P \leq Q$ since $Q \in ord(\emptyset, P)$.

For the induction case, assume $CT(\emptyset, Q, Q) = C \texttt{ ext } T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}$ and $T'\ m(\vec{T}\ \vec{x})\ \{e\} \notin \overline{M}$. and $(\emptyset, P, Q) = Q'$. and $mbody(P, Q', m) = (\vec{x}, e)$. By the induction hypothesis, there is a P' not before Q' in $ord(\emptyset, P)$ where $\vdash P \leq P'$ such that $\texttt{this} : P', \vec{x} : \vec{T} \vdash e : T'$. Since P' follows Q in $ord(\emptyset, P)$, the case holds. □

The field conformance lemma and its proof are similar to the method conformance lemma.

Lemma A.3 (Field conformance) If $ftype(\emptyset, P, f) = T$, and $finit(\emptyset, P, f) = e$, then there is a $P' \in ord(\emptyset, P)$ where $\vdash P \leq P'$ such that $\texttt{this} : P' \vdash e : T$.

Proof. Similar to the proof of Lemma A.2. □

A.3 Type equivalence and subtyping

Dependent classes and prefix types permit two types to contain exactly the same sets of values. For instance the types $\ell_{\mathtt{A}'}.\mathtt{class}$ and $\mathtt{A}[\ell'_{\mathtt{A}'.\mathtt{B}}.\mathtt{class}:\mathtt{A.B}]$ may each contain only values of run-time class $\mathtt{A}'$. We consider these types to be equivalent as follows.

Definition A.4 Two types T and T' are *equivalent*, written $T \approx T'$, if they are related by the smallest reflexive, symmetric, transitive closure of the following rules:

$$\frac{T \approx T'}{T.C \approx T'.C}$$

$$\frac{\mathit{exact\text{-}class}(T) = P \quad \mathit{exact\text{-}class}(T') = P}{T \approx T'}$$

$$\frac{T \approx T'}{P[T:P.C] \approx P[T':P.C]}$$

We extend the definition of equivalent types to class declarations and sets of class declarations.

Definition A.5 We write $L \approx L'$ if $L = C\ \mathtt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}$ and $L' = C\ \mathtt{ext}\ T'_s\ \{\overline{L'}\ \vec{F}\ \overline{M}\}$ and $T_s \approx T'_s$ and $\overline{L} \approx \overline{L'}$, where $\overline{L} \approx \overline{L'}$ if $L \in \overline{L}$ if and only if there is an $L' \in \overline{L'}$ such that $L \approx L'$.

Next, we prove several preliminary facts about equivalent types, equivalent classes and subtyping, which are needed for the subject reduction proof.

Lemma A.6 If $\overline{L_1} \approx \overline{L'_1}$, then $\overline{L_1} \bullet \overline{L_2} \approx \overline{L'_1} \bullet \overline{L'_2}$.

Proof. By induction of the member class inheritance rules in Figure 10. □

Lemma A.7 If $T_1 \approx T'_1$ and $T_2 \approx T'_2$, then

(i) $\mathit{classes}(\Gamma, T_2, T_1) \approx \mathit{classes}(\Gamma, T'_2, T'_1)$.

(ii) $\mathit{super}(\Gamma, T_1) \approx \mathit{super}(\Gamma, T'_1)$.

(iii) $\mathit{fields}(\Gamma, T_1, T_1) = \mathit{fields}(\Gamma, T'_1, T'_1)$.

Proof. The proof is by induction on the derivation of $CT(\Gamma, T_2, T_1)$.

Case CT-OUTER:

Then $T_1 = C$ and $T'_1 = C$.

By CT-OUTER, if $CT(\Gamma, T_2, C) = C\ \mathtt{ext}\ P\ \{\overline{L_s} \bullet \overline{L}\{T_2/\mathtt{This}\}\ \vec{F}\ \overline{M}\}$, then $TCT(C) = C\ \mathtt{ext}\ P\ \{\overline{L}\ \vec{F}\ \overline{M}\}$ and $\mathit{classes}(\Gamma, T_2, P) = \overline{L_s}$.

By the induction hypothesis, $\mathit{classes}(\Gamma, T'_2, P) = \overline{L'_s}$, where $\overline{L'_s} \approx \overline{L_s}$.

Thus, we can derive $CT(\Gamma, T'_2, C) = C\ \mathtt{ext}\ P\ \{\overline{L'_s} \bullet \overline{L}\{T'_2/\mathtt{This}\}\ \vec{F}\ \overline{M}\}$.

By Lemma A.6, $\overline{L_s} \bullet \overline{L}\{T_2/\mathtt{This}\} \approx \overline{L'_s} \bullet \overline{L}\{T'_2/\mathtt{This}\}$.

(i), (ii), and (iii) easily follow.

Case CT-NEST:

Then $T_1 = T_0.C$ and $T'_1 = T'_0.C$, where $T_0 \approx T'_0$.

By CT-NEST $CT(\Gamma, T_2, T_0.C) = C\ \mathtt{ext}\ T_s\ \{\overline{L_s} \bullet \overline{L}\{T_2/\mathtt{This}\}\ \vec{F}\ \overline{M}\}$, where $C\ \mathtt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\} \in \mathit{classes}(\Gamma, T_0, T_0)$, and $\mathit{classes}(\Gamma, T_2, T_s) = \overline{L_s}$.

By the induction hypothesis, $\mathit{classes}(\Gamma, T_0, T_0) \approx \mathit{classes}(\Gamma, T'_0, T'_0)$. Thus, there is a declaration $C\ \mathtt{ext}\ T'_s\ \{\overline{L'}\ \vec{F}\ \overline{M}\} \in \mathit{classes}(\Gamma, T'_0, T'_0)$ such that $T_s \approx T'_s$ and $\overline{L} \approx \overline{L'}$. Also by the induction hypothesis, $\mathit{classes}(\Gamma, T_2, T_s) \approx \mathit{classes}(\Gamma, T'_2, T'_s)$. Thus, by CT-NEST, we can derive $CT(\Gamma, T'_2, T'_0.C) = C\ \mathtt{ext}\ T'_s\ \{\overline{L'_s} \bullet \overline{L'}\{T'_2/\mathtt{This}\}\ \vec{F}\ \overline{M}\}$,

By Lemma A.6, $\overline{L_s} \bullet \overline{L}\{T_2/\mathtt{This}\} \approx \overline{L'_s} \bullet \overline{L}\{T'_2/\mathtt{This}\}$.

(i), (ii), and (iii) easily follow.

Case CT-RUNTIME:

Then there is a P such that $\mathit{exact\text{-}class}(T_1) = P = \mathit{exact\text{-}class}(T_2)$. By CT-RUNTIME, $CT(\Gamma, T_2, T_1) = \_\ \texttt{ext}\ P\ \{\overline{L} \bullet \emptyset\}$ where $\mathit{classes}(\Gamma, T_2, P) = \overline{L}$. By the induction hypothesis, $\mathit{classes}(\Gamma, T_2', P) = \overline{L'} \approx \overline{L}$. Thus, we can derive $CT(\Gamma, T_2', T_1') = \_\ \texttt{ext}\ P\ \{\overline{L'} \bullet \emptyset\}$.

By Lemma A.6, $\overline{L} \bullet \emptyset \approx \overline{L'} \bullet \emptyset$.

(i), (ii), and (iii) easily follow.

Case CT-DEP:

Then $T_1 = p.\texttt{class}$ and $T_1' = p'.\texttt{class}$.

Since $\Gamma \vdash p\ \texttt{final}\ P$, $\Gamma \vdash p'\ \texttt{final}\ P$, By CT-DEP, $CT(\Gamma, T_2, T_1) = \_\ \texttt{ext}\ P\ \{\overline{L} \bullet \emptyset\}$ where $\mathit{classes}(\Gamma, T_2, P) = \overline{L}$. By the induction hypothesis, $\mathit{classes}(\Gamma, T_2', P) = \overline{L'} \approx \overline{L}$. Thus, we can derive $CT(\Gamma, T_2', T_1') = \_\ \texttt{ext}\ P\ \{\overline{L'} \bullet \emptyset\}$.

By Lemma A.6, $\overline{L} \bullet \emptyset \approx \overline{L'} \bullet \emptyset$.

(i), (ii), and (iii) easily follow.

Case CT-PRE:

Then $T_1 = P\texttt{[}T_{1x}\texttt{:}P.C\texttt{]}$ and $T_1' = P\texttt{[}T_{1x}'\texttt{:}P.C\texttt{]}$ where $T_{1x} \approx T_{1x}'$.

By CT-PRE, $CT(\Gamma, T_2, T_1) = \_\ \texttt{ext}\ P\ \{\overline{L} \bullet \emptyset\}$ where $\mathit{classes}(\Gamma, T_2, P) = \overline{L}$. By the induction hypothesis, $\mathit{classes}(\Gamma, T_2', P) = \overline{L'} \approx \overline{L}$. Thus, we can derive $CT(\Gamma, T_2', T_1') = \_\ \texttt{ext}\ P\ \{\overline{L'} \bullet \emptyset\}$.

By Lemma A.6, $\overline{L} \bullet \emptyset \approx \overline{L'} \bullet \emptyset$.

(i), (ii), and (iii) easily follow.

□

Lemma A.8 If $T_1 \approx T_1'$ and $T_2 \in \mathit{ord}(\Gamma, T_1)$, then there is a T_2' such that $T_2 \approx T_2'$ and $T_2' \in \mathit{ord}(\Gamma, T_1')$.

Proof. The proof is by induction on the definition of $\mathit{ord}(\Gamma, T_1)$.

Case $T_1 = \texttt{Object}$:

Then $T_2 = \texttt{Object}$ and $T_2 \in \mathit{ord}(\Gamma, T_1')$ trivially.

Case $T_1 = T_{1x}.C$:

Then $T_1' = T_{1x}'.C$.

If $T_2 \in \mathit{ord}(\Gamma, T_{1x}).C$, then $T_2 = T_{2x}.C$ for some T_{2x} and $\Gamma \vdash T_{2x}.C$ wf and $T_{2x} \in \mathit{ord}(\Gamma, T_{1x})$. Therefore, $T_2' = T_{2x}'.C$ and, by the induction hypothesis, $T_{2x}' \in \mathit{ord}(\Gamma, T_{1x}')$. Thus, $T_2' \in \mathit{ord}(\Gamma, T_1')$.

Otherwise, $T_2 \in \mathit{ord}(\Gamma, \mathit{super}(\Gamma, T_1))$. By Lemma A.7, $\mathit{super}(\Gamma, T_1) \approx \mathit{super}(\Gamma, T_1')$, and by the induction hypothesis $T_2' \in \mathit{ord}(\Gamma, \mathit{super}(\Gamma, T_1'))$.

Otherwise:

If $T_1 = T_2$, then take $T_2' = T_1'$.

Otherwise, $T_2 \in \mathit{ord}(\Gamma, \mathit{super}(\Gamma, T_1))$. By Lemma A.7, $\mathit{super}(\Gamma, T_1) \approx \mathit{super}(\Gamma, T_1')$, and by the induction hypothesis $T_2' \in \mathit{ord}(\Gamma, \mathit{super}(\Gamma, T_1'))$.

□

Lemma A.9 If $\Gamma \vdash T_1' \leq T_1$, then there is a $T_2 \approx T_1$ such that $T_2 \in \mathit{ord}(\Gamma, T_1')$. Moreover, if $T_1' = P_1'$ and $T_1 = P_1$, then $T_2 = T_1 = P_1$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash T_1' \leq T_1$. The reflexive case holds trivially. The transitive case holds by the induction hypothesis.

Case $\leq$-EXTENDS:

Then $\mathit{super}(\Gamma, T_1') = T_1$ and $T_1 \in \mathit{ord}(\Gamma, T_1')$ trivially. If $T_1' = P_1'$ and $T_1 = P_1$, then $T_2 = T_1 = P_1$ trivially.

Case $\leq$-NEST:

Then $T_1 = T.C$ and $T_1' = T'.C$ where $\Gamma \vdash T' \leq T$.

By the induction hypothesis there is a $T'' \in \mathit{ord}(\Gamma, T')$ such that $T'' \approx T$. Thus, $T_2 = T''.C$ and $T_2 \in \mathit{ord}(\Gamma, T').C$. and hence $T_2 \in \mathit{ord}(\Gamma, T_1')$.

If $T_1' = P'.C$ and $T_1 = P.C$, then by the induction hypothesis, $T'' = T = P$ and thus $T_2 = P.C = T_1$.

Case $\leq$-RUNTIME:

Then $T_1' \approx T_1$, and the first part of the lemma holds by Lemma A.8. In this case, T_1 and T_1' cannot be simple non-dependent class types so the second part holds vacuously.

□

Lemma A.10 If $\Gamma \vdash T_1' \leq T_1$ and $T_2 \in \mathit{ord}(\Gamma, T_1)$, then there is a T_2' such that $T_2 \approx T_2'$ and $T_2' \in \mathit{ord}(\Gamma, T_1')$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash T_1' \leq T_1$. The reflexive case holds trivially. The transitive case holds by the induction hypothesis.

Case $\leq$-EXTENDS:

Then $CT(\Gamma, T_1', T_1') = C$ `ext` T_1 $\{\overline{L}\ \vec{F}\ \overline{M}\}$.

If $T_2 \in \mathit{ord}(\Gamma, T_1)$, then $T_2 \in \mathit{ord}(\Gamma, T_1')$ since $T_1' \neq$ `Object` and $\mathit{ord}(\Gamma, T_1')$ contains $\mathit{ord}(\Gamma, \mathit{super}(\Gamma, T_1'))$.

Case $\leq$-NEST:

Then $T_1 = T.C$ and $T_1' = T'.C$ where $\Gamma \vdash T' \leq T$.

Suppose $T_2 \in \mathit{ord}(\Gamma, T_1)$. Either $T_2 \in \mathit{ord}(\Gamma, T).C$ or $T_2 \in \mathit{ord}(\Gamma, \mathit{super}(\Gamma, T.C))$.

If $T_2 \in \mathit{ord}(\Gamma, T).C$, then $T_2 = T_3.C$ for some $T_3 \in \mathit{ord}(\Gamma, T)$. By the inductive hypothesis, there is a T_3' such that $T_3' \in \mathit{ord}(\Gamma, T')$, and so $T_2' = T_3'.C \in \mathit{ord}(\Gamma, T').C$ and so $T_2' \in \mathit{ord}(\Gamma, T_1')$.

Otherwise, $T_2 \in \mathit{ord}(\Gamma, \mathit{super}(\Gamma, T_1))$. By Lemma A.9, there is a $T_3 \approx T_1$ such that $T_3 \in \mathit{ord}(\Gamma, T_1')$. Therefore, there is a type $T \approx \mathit{super}(\Gamma, T_3) \approx \mathit{super}(\Gamma, T_1)$ such that $T \in \mathit{ord}(\Gamma, T_1')$. By Lemma A.8, there is a $T_2' \approx T_2$ in $\mathit{ord}(\Gamma, T)$. But by the definition of *ord*, every element of $\mathit{ord}(\Gamma, T)$ is in $\mathit{ord}(\Gamma, T_1')$, so $T_2' \in \mathit{ord}(\Gamma, T_1')$.

Case $\leq$-RUNTIME:

Then there is a P such that $\mathit{exact\text{-}class}(T_1) = P$ and $\mathit{exact\text{-}class}(T_1') = P$. The case holds by Lemma A.8.

□

Lemma A.11 If $\Gamma \vdash T' \leq T$, and $\mathit{ftype}(\Gamma, T, f_i) = [\texttt{final}]\ T_i$, then $\mathit{ftype}(\Gamma, T', f_i) = [\texttt{final}]\ T_i$.

Proof. Assume $\Gamma \vdash T' \leq T$, and $\mathit{ftype}(\Gamma, T, f_i) = [\texttt{final}]\ T_i$. Then it must be the case that for some type $T_d \in \mathit{ord}(\Gamma, T)$, the judgment $CT(\Gamma, T, T_d) = C_d$ `ext` T_{sd} $\{\overline{L_d}\ \vec{F_d}\ \overline{M_d}\}$ occurs in the derivation of $\mathit{ftype}(\Gamma, T, f_i) = \texttt{final}\ T_i$, and `final` T_i $f_i = e_i$ is in $\vec{F_d}$.

By Lemma A.10, there is a T_d' such that $T_d \approx T_d'$ and $T_d' \in \mathit{ord}(\Gamma, T')$. By Lemma A.7, $\mathit{ftype}(\Gamma, T_d', f_i) = [\texttt{final}]\ T_i$, and since $T_d' \in \mathit{ord}(\Gamma, T')$, we have $\mathit{ftype}(\Gamma, T', f_i) = [\texttt{final}]\ T_i$, as required. □

We also define a weaker notion of set inclusion for sets of class declarations, which is needed for certain substitution results. Informally, we write $\overline{L_1} \Subset \overline{L_2}$ if for every class C that is declared in $\overline{L_1}$, there is a class by the same name declared in $\overline{L_2}$.

Definition A.12 We write $\overline{L_1} \Subset \overline{L_2}$ if for each $L_1 = C$ `ext` T_1 $\{\overline{L_1'}\ \vec{F}_1\ \overline{M_1}\}$ in $\overline{L_1}$ there is an $L_2 = C$ `ext` T_2 $\{\overline{L_2'}\ \vec{F}_2\ \overline{M_2}\}$ in $\overline{L_2}$ such that $\overline{L_1'} \Subset \overline{L_2'}$.

The next two lemmas state that the $\Subset$ relation is closed under member class inheritance, and that the set of member classes is covariant with respect to subtyping.

Lemma A.13 If $\overline{L_1} \Subset \overline{L_1'}$ and $\overline{L_2} \Subset \overline{L_2'}$, then $\overline{L_1} \bullet \overline{L_2} \Subset \overline{L_1'} \bullet \overline{L_2'}$.

Proof. By induction of the member class inheritance rules in Figure 10. □

The following lemma states that if a type has a nested class C, then its subtypes also have a nested class C.

Lemma A.14 If $\Gamma \vdash T' \leq T$, and $\mathit{classes}(\Gamma, T_0, T) = \overline{L}$, and $\mathit{classes}(\Gamma, T_0, T') = \overline{L'}$, then $\overline{L} \Subset \overline{L'}$.

Proof. By induction on the derivation of $\Gamma \vdash T' \leq T$. The reflexive case holds trivially. The transitive case holds by the induction hypothesis.

Assume $\mathit{classes}(\Gamma, T_0, T) = \overline{L}$ and $\mathit{classes}(\Gamma, T_0, T') = \overline{L}'$ and $\Gamma \vdash T' \leq T$

Case $\leq$-EXTENDS:

$\mathit{super}(\Gamma, T') = T$, that is $CT(\Gamma, T', T') = C$ `ext` T $\{\overline{L}\ \vec{F}\ \overline{M}\}$. Lemma A.13 and inspection of all of the class table rules shows that $\overline{L} \Subset \overline{L'}$.

Case $\leq$-NEST:

$T = T_1.C$ and $T' = T_1'.C$ and $\Gamma \vdash T_1' \leq T_1$.

Let $\mathit{classes}(\Gamma, T_1, T_1) = \overline{L_1}$ and $\mathit{classes}(\Gamma, T_1', T_1') = \overline{L_1'}$. By CT-NEST, if $CT(\Gamma, T_0, T) = C$ `ext` T_s $\{\overline{L_C}\ \vec{F_C}\ \overline{M_C}\}$, then C `ext` T_s $\{\overline{L_C}\ \vec{F_C}\ \overline{M_C}\} \in \overline{L_1}$. Note that $\overline{L} = \overline{L_s} \bullet \overline{L_C}\{T_0/\texttt{This}\}$, where $\mathit{classes}(\Gamma, T_0, T_s) = \overline{L_s}$.

By the inductive hypothesis, $\overline{L_1} \Subset \overline{L_1'}$, and so there is some class C `ext` T_s' $\{\overline{L_C'}\ \vec{F_C'}\ \overline{M_C'}\} \in \overline{L_1'}$ such that $\overline{L_C} \Subset \overline{L_C'}$. Note that $\overline{L'} = \overline{L_s'} \bullet \overline{L_C'}\{T_0/\texttt{This}\}$, where $\mathit{classes}(\Gamma, T_0, T_s') = \overline{L_s'}$.

Now, $\overline{L_s} \bullet \overline{L_C}\{T_0/\texttt{This}\} \Subset \overline{L_s'} \bullet \overline{L_C'}\{T_0/\texttt{This}\}$ and $\overline{L_C} \Subset \overline{L_C'}$, and so $\overline{L} \Subset \overline{L'}$.

Case $\leq$-RUNTIME:

$\mathit{exact\text{-}class}(T) = \mathit{exact\text{-}class}(T')$, and so by CT-RUNTIME, $\mathit{super}(\Gamma, T) = \mathit{super}(\Gamma, T')$. It is thus easy to see that $\mathit{classes}(\Gamma, T_0, T) \Subset \mathit{classes}(\Gamma, T_0, T')$ (and also $\mathit{classes}(\Gamma, T_0, T') \Subset \mathit{classes}(\Gamma, T_0, T)$).

□

A.4 Type schemas

During class lookup, any occurrence of the type schema `This` appearing in an `extends` clause of a nested class will be substituted with a subtype of the enclosing class. The resulting type should be well-formed. The following two lemmas are needed to prove this.

Lemma A.15 If $\Gamma \vdash T_0 \leq P$, and $\mathrm{loc}(T_0) = \emptyset$, and $\texttt{this}:P \in \Gamma$, then $\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\} \leq Q$ implies $\Gamma \vdash S\{T_0/\texttt{This}\} \leq Q$.

Proof. By structural induction on S. Assume $\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\} \leq Q$.

Case $S = D$:

Trivial.

Case $S = S'.D$:

Follows from induction hypothesis and $\leq$-NEST.

Case $S = \texttt{This}$:

Then $S\{\texttt{this.class}/\texttt{This}\} = \texttt{this.class}$ and $S\{T_0/\texttt{This}\} = T_0$. If $\Gamma \vdash \texttt{this.class} \leq Q$, then $\Gamma \vdash P \leq Q$. Thus, $\Gamma \vdash T_0 \leq P$ implies by transitivity $\Gamma \vdash T_0 \leq Q$.

Case $S = P'[S':P'.C]$:

If S' is `This`-free, then $S\{\texttt{this.class}/\texttt{This}\} = S\{T_0/\texttt{This}\}$ and the case holds trivially.

Since $\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\} \leq Q$, we have $\Gamma \vdash P' \leq Q$.

If *is-exact*(T_0), then if $\Gamma \vdash S\{T_0/\texttt{This}\} \leq P'$ by $\leq$-EXTENDS and CT-PRE. By transitivity, $\Gamma \vdash S\{T_0/\texttt{This}\} \leq Q$.

Since $\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\}$ wf, we have $\Gamma \vdash S'\{\texttt{this.class}/\texttt{This}\} \leq P'.C$.

By the induction hypothesis, $\Gamma \vdash S'\{T_0/\texttt{This}\} \leq P'.C$. Thus, there is a type of the form $P''.C$ in $ord(\Gamma, S'\{T_0/\texttt{This}\})$ where $\Gamma \vdash P'' \leq P'$. Thus, $S\{T_0/\texttt{This}\} = prefix(P', S'\{T_0/\texttt{This}\}, S'\{T_0/\texttt{This}\}, P'.C) = P''$ exists. By transitivity, since $\Gamma \vdash P'' \leq P'$, $\Gamma \vdash P'' \leq Q$.

□

Lemma A.16 Assume $\Gamma \vdash T_0 \leq P$ and $\texttt{this} : P \in \Gamma$, and $classes(\Gamma, S\{\texttt{this.class}/\texttt{This}\}, S\{\texttt{this.class}/\texttt{This}\}) = \overline{L}$ and $classes(\Gamma, S\{T_0/\texttt{This}\}, S\{T_0/\texttt{This}\}) = \overline{L'}$. Then $\overline{L} \Subset \overline{L'}$. That is, if there is a class declaration for C in $\overline{L}$, there is also a class declaration for C in $\overline{L'}$.

Proof. Follows from Lemma A.15, and Lemma A.14. □

Now, we can show that a type produced by substituting for a type schema is well-formed.

Lemma A.17 If $\texttt{this} : P \in \Gamma$ and $\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\}$ wf, then for any T_0 such that for any $\Gamma \vdash T_0$ wf and $\Gamma \vdash T_0 \leq P$, $\Gamma \vdash S\{T_0/\texttt{This}\}$ wf.

Proof. By structural induction on S.

Case $S = C$:

Trivial since $S\{\texttt{this.class}/\texttt{This}\} = S\{T_0/\texttt{This}\}$.

Case $S = S'.C$:

Then $S\{T_0/\texttt{This}\} = S'\{T_0/\texttt{This}\}.C$. By the induction hypothesis, $\Gamma \vdash S'\{T_0/\texttt{This}\}$ wf.

$\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\}$ wf, there is a class declaration for C in $classes(\Gamma, S'\{\texttt{this.class}/\texttt{This}\}, S'\{\texttt{this.class}/\texttt{This}\})$.

By Lemma A.16, there is a class declaration for C in $classes(\Gamma, S'\{T_0/\texttt{This}\}, S'\{T_0/\texttt{This}\})$.

Thus, by WF-NEST, $\Gamma \vdash S\{T_0/\texttt{This}\}$ wf.

Case $S = \texttt{This}$:

Trivial since $S\{T_0/\texttt{This}\} = T_0$.

Case $S = Q[S' : Q.C]$:

Then $S\{T_0/\texttt{This}\} = Q[S'\{T_0/\texttt{This}\} : Q.C]$ By the induction hypothesis, $\Gamma \vdash S'\{T_0/\texttt{This}\}$ wf.

Let $T' = S'\{T_0/\texttt{This}\}$. There are two cases.

- If $T' = Q'$ for some Q', then $S\{T_0/\texttt{This}\} = prefix(Q, Q', Q', Q.C)$. We need to show that this type exists and is well-formed.
  Since $\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\}$ wf, by WF-PRE, $\Gamma \vdash S'\{\texttt{this.class}/\texttt{This}\} \leq Q.C$; and therefore by Lemma A.15, $\Gamma \vdash Q' \leq Q.C$.
  By Lemma A.10, $Q.C \in ord(\Gamma, Q')$; therefore $prefix(Q, Q', Q', Q.C)$ exists and is well-formed.
- Otherwise, since $T' \neq Q'$ for any Q', we have $S\{T_0/\texttt{This}\} = Q[T' : Q.C]$.
  Since $\Gamma \vdash S\{\texttt{this.class}/\texttt{This}\}$ wf, by WF-PRE, $\Gamma \vdash S'\{\texttt{this.class}/\texttt{This}\} \leq Q.C$; and therefore by Lemma A.15, $\Gamma \vdash T' \leq Q.C$.
  It then follows from WF-PRE that $\Gamma \vdash S\{T_0/\texttt{This}\}$ wf.

□

We can then show that the *super* function returns a well-formed type; and from this lemma we can conclude that if a type is well-formed, all of its supertypes are well-formed also.

Lemma A.18 If $\Gamma \vdash T$ wf and $\mathit{super}(\Gamma, T) = T_s$, then $\Gamma \vdash T_s$ wf.

Proof. By Lemma A.1, each class declaration in TCT^*, including those of nested classes, is checked with OK-CLASS. This rule requires that if C `ext` S $\{\overline{L}\ \vec{F}\ \overline{M}\}$ is a class declaration in the body of P, then $\texttt{this}:P \vdash S\{\texttt{this.class}/\texttt{This}\}$ wf. It follows immediately from Lemma A.17 that any class declaration returned by CT has a well-formed immediate superclass. □

Lemma A.19 If $\Gamma \vdash T$ wf and if $\Gamma \vdash T \leq T'$, then $\Gamma \vdash T'$ wf.

Proof. By induction on the derivation of $\Gamma \vdash T \leq T'$, using Lemma A.18 for the $\leq$-EXTENDS case. □

A.5 Typing environments

To prove subject reduction, we first need to prove several substitution lemmas. Since we have dependent types, variables that appear in types, as well as those in expressions, may be substituted. We define substitution on well-formed typing environments Γ as follows.

Definition A.20

$$\begin{aligned} \emptyset\{b/x\} &= \emptyset \\ (\Gamma, x:T)\{b/x\} &= \Gamma \\ (\Gamma, y:T)\{b/x\} &= \Gamma\{b/x\}, y:T\{b/x\} \end{aligned}$$

We also introduce a few lemmas that will be useful for proving subject reduction.

Lemma A.21 (Weakening) If $x \notin \mathrm{dom}(\Gamma)$, then for any T', if $\Gamma \vdash e:T$, then $\Gamma, x:T' \vdash e:T$.

Proof. Simple proof by induction on the derivation of $\Gamma \vdash e:T$. □

Lemma A.22 (Path weakening) If $x \notin \mathrm{dom}(\Gamma)$, then for any T', if $\Gamma \vdash p$ `final` T, then $\Gamma, x:T' \vdash p$ `final` T.

Proof. Simple proof by induction on the derivation of $\Gamma \vdash p$ `final` T. □

Lemma A.23 (Subtyping weakening) If $x \notin \mathrm{dom}(\Gamma)$, then for any T', if $\Gamma \vdash T_1 \leq T_2$, then $\Gamma, x:T' \vdash T_1 \leq T_2$.

Proof. Simple proof by induction on the derivation of $\Gamma \vdash T_1 \leq T_2$. □

A.6 Substitution

Next, we prove several substitution lemmas. Because the language includes dependent classes and prefix types, the proof requires more complex substitution lemmas than the proof of soundness for FJ in [15]. There is a substitution lemma for most of the judgments in the semantics.

The next few lemmas are preliminaries to the substitution lemmas. They state some useful properties about types in empty type environments.

Lemma A.24 For a base value b, if $\vdash b:T$ then $\vdash T$ wf.

Proof. Consider the derivation of $\vdash b:T$. The last rule used in such a derivation is one of T-LOC, T-NULL or T-$\leq$. The first two of these rules require that $\vdash T$ wf. For the last rule, T-$\leq$, well-formedness follows from Lemma A.19. □

Lemma A.25 If $\vdash T$ wf and $\mathit{exact\text{-}class}(T) = P$, then $\mathit{super}(\emptyset, T) = P$.

Proof. Trivial by examination of $\leq$-EXTENDS and CT-RUNTIME. □

Lemma A.26 If $\vdash P$ wf and $\vdash Q$ wf, and $Q \in \mathit{ord}(\emptyset, P)$, then $\vdash P \leq Q$.

Proof. Assume $Q \in \mathit{ord}(\emptyset, P)$. The proof is by induction on the definition of $Q \in \mathit{ord}(\emptyset, P)$.

Case $P = \texttt{Object}$:

Trivial since $\mathit{ord}(\emptyset, \texttt{Object}) = [\texttt{Object}]$.

Case $P = P'.C$:

Then $ord(\emptyset, P) = ord(\emptyset, P').C, ord(\emptyset, super(\emptyset, P))$.

There are two cases:

If $Q \in ord(\emptyset, P').C$, then $Q = Q'.C$ and $Q' \in ord(\emptyset, P')$. By the induction hypothesis, $\vdash P' \leq Q'$, and by $\leq$-NEST, $\vdash P'.C \leq Q'.C$, or equivalently $\vdash P \leq Q$.

If $Q \in ord(\emptyset, super(\emptyset, P))$, then by the induction hypothesis, $\vdash super(\emptyset, P) \leq Q$, and thus $\vdash P \leq Q$ by $\leq$-EXTENDS and transitivity.

Otherwise:

Then $ord(\emptyset, P) = P, ord(\emptyset, super(\emptyset, P))$.

If Q is in $ord(\emptyset, P)$, then either $Q = P$ and the case holds by reflexivity; or, $Q \in ord(\emptyset, super(\emptyset, P))$. By the induction hypothesis, $\vdash super(\emptyset, P) \leq Q$, and thus $\vdash P \leq Q$ by $\leq$-EXTENDS and transitivity.

□

Lemma A.27 If $T = Q[T' : Q.C]$, $\vdash T$ wf, and $exact\text{-}class(T) = P$, then $\vdash P \leq Q$.

Proof. Assume $T = Q[T' : Q.C]$. Since $\vdash T$ wf, $\vdash T' \leq Q.C$. Then $exact\text{-}class(T) = prefix(Q, exact\text{-}class(T'), exact\text{-}class(T'), Q.C)$ where $exact\text{-}class(T') = P'$.

Since T is well-formed, by WF-PRE, T' is well-formed and $\vdash T' \leq Q.C$. By Lemma A.25, $super(\emptyset, T') = P'$. Since $is\text{-}exact(T')$, $\leq$-NEST cannot apply, so for $\vdash T' \leq Q.C$, it must be that $\vdash P' \leq Q.C$. Moreover, by Lemma A.9, $Q.C$ is in $ord(\emptyset, P')$.

By the definition of *prefix*, $P.C$ is the first type of the form $Q'.C$ in $ord(\emptyset, P')$. Thus, since $Q.C \in ord(\emptyset, P')$, either $P = Q$, or $P.C$ occurs before $Q.C$ in $ord(\emptyset, P')$. If the former, then $\vdash P \leq Q$ by reflexivity. If the latter, than there is a type T'' such that $P.C$ occurs before $Q.C$ in $ord(\emptyset, T'').C$. Therefore, P occurs before Q in $ord(\emptyset, T'')$. Since $P.C$ is the first type of the form $Q'.C$ in $ord(\emptyset, P')$, it must be that P is the first element of $ord(\emptyset, T'')$. Thus $T'' = P$, and therefore $\vdash P \leq Q$. □

The following two lemmas show that class table rules and subtyping judgments for simple classes do not depend on the variable context.

Lemma A.28 If $CT(\Gamma, P_0, P) = C \texttt{ ext } T_s \{\overline{L}\ \vec{F}\ \overline{M}\}$ then $CT(\emptyset, P_0, P) = C \texttt{ ext } T_s \{\overline{L}\ \vec{F}\ \overline{M}\}$.

Proof. By induction on $CT(\Gamma, P_0, P) = C \texttt{ ext } T_s \{\overline{L}\ \vec{F}\ \overline{M}\}$. The inductive hypothesis is that for all $CT(\Gamma, P_0', P')$ occurring in the derivation of $CT(\Gamma, P_0, P)$, we have $CT(\emptyset, P_0', P')$, and moreover, that if $C' \texttt{ ext } T_s' \{\overline{L'}\ \vec{F'}\ \overline{M'}\} \in classes(\Gamma, P_0, P)$, then T_s' is a simple class.

Case CT-OUTER, CT-NEST:

$CT(\emptyset, P_0, P)$ follows from inductive hypothesis; for all $C' \texttt{ ext } T_s' \{\overline{L'}\ \vec{F'}\ \overline{M'}\} \in \overline{L_s} \bullet \overline{L}\{P_0/\texttt{This}\}$, we must have T_s' a simple class, because of the inductive hypothesis, and the fact that a simple class P_0 is substituted for `This`.

Case CT-RUNTIME, CT-DEP, CT-PRE:

Impossible.

□

Lemma A.29 If $\Gamma \vdash P \leq P'$ then $\emptyset \vdash P \leq P'$

Proof. Proof is by induction on $\Gamma \vdash P \leq P'$.

Case $\leq$-EXTENDS:

$super(\Gamma, P) = P'$. Result follows from Lemma A.28.

Case $\leq$-NEST:

Follows from inductive hypothesis.

Case $\leq$-RUNTIME:

No simple class is in dom(*exact-class*), and so this case is impossible.

□

The next lemma states that the *ord* function is closed under substitution.

Lemma A.30 If $x : T_x \in \Gamma$ and $\vdash b : T_x$, and $T_0 \in ord(\Gamma, T_1)$ and $\Gamma\{b/x\} \vdash super(\Gamma\{b/x\}, T_1\{b/x\}) \leq (super(\Gamma, T_1))\{b/x\}$, then there is a T such that $T \approx T_0\{b/x\}$ and $T \in ord(\Gamma\{b/x\}, T_1\{b/x\})$.

Proof. Assume $x : T_x \in \Gamma$ and $\vdash b : T_x$, and $\Gamma\{b/x\} \vdash super(\Gamma\{b/x\}, T_1\{b/x\}) \leq (super(\Gamma, T_1))\{b/x\}$. Proof is by induction of the definition of $ord(\Gamma, T_1)$. Inductive hypothesis is that for all $T_0 \in ord(\Gamma, T_1)$, there is a T such that $T \approx T_0\{b/x\}$ and $T \in ord(\Gamma\{b/x\}, T_1\{b/x\})$.

Case $T_1 = \texttt{Object}$:

If $T_0 \in ord(\Gamma, T_1)$, then it must be the case that $T_0 = \texttt{Object}$, and so $T_0\{b/x\} = \texttt{Object} \in [\texttt{Object}] = ord(\Gamma\{b/x\}, T_1\{b/x\})$.

Case $T_1 = T_1'.C$:

$ord(\Gamma, T_1) = ord(\Gamma, T_1').C$, $ord(\Gamma, super(\Gamma, T_1'.C))$, so either $T_0 = T_0'.C$ and $T_0' \in ord(\Gamma, T_1')$, or $T_0 \in ord(\Gamma, super(\Gamma, T_1))$.

If $T_0 = T_0'.C$ and $T_0' \in ord(\Gamma, T_1')$, then by the inductive hypothesis there is a T' such that $T' \approx T_0'\{b/x\}$ and $T' \in ord(\Gamma\{b/x\}, T_1'\{b/x\})$. Now, $T'.C \approx T_0\{b/x\}$ and $T'.C \in ord(\Gamma\{b/x\}, T_1'\{b/x\}).C = ord(\Gamma\{b/x\}, T_1\{b/x\})$, as required.

If $T_0 \in ord(\Gamma, super(\Gamma, T_1))$, then by the inductive hypothesis, there is a T' such that $T' \approx T_0\{b/x\}$ and $T' \in ord(\Gamma\{b/x\}, super(\Gamma, T_1)\{b/x\})$. By Lemma A.10, there is a T such that $T' \approx T$ and $T \in ord(\Gamma\{b/x\}, super(\Gamma\{b/x\}, T_1\{b/x\}))$. By transitivity, $T \approx T_0\{b/x\}$, as required.

Case $T_1 \neq T.C$ and $T_1 \neq \texttt{Object}$:

$ord(\Gamma, T_1) = T_1, ord(\Gamma, super(\Gamma, T_1))$, so either $T_0 = T_1$, or $T_0 \in ord(\Gamma, super(\Gamma, T_1))$.

If $T_0 = T_1$, then $T_0\{b/x\} = T_1\{b/x\}$ and so $T_0\{b/x\} \in ord(\Gamma\{b/x\}, T_1\{b/x\}) = T_1\{b/x\}, ord(\Gamma, super(\Gamma, T_1\{b/x\}))$, as required.

If $T_0 \in ord(\Gamma, super(\Gamma, T_1))$, then by the inductive hypothesis, there is a T' such that $T' \approx T_0\{b/x\}$, and $T' \in ord(\Gamma\{b/x\}, super(\Gamma, T_1)\{b/x\})$. By Lemma A.10, there is a T such that $T' \approx T$ and $T \in ord(\Gamma\{b/x\}, super(\Gamma\{b/x\}, T_1\{b/x\}))$. By transitivity, $T \approx T_0\{b/x\}$, as required.

□

The *exact-class* function is also closed under substitution.

Lemma A.31 Assume $x : T_x \in \Gamma$ and $\vdash b : T_x$. If $T_0 \approx T_1$, then $T_0\{b/x\} \approx T_1\{b/x\}$, and if $\textit{exact-class}(T_0) = P$ then $\textit{exact-class}(T_0\{b/x\}) = P$.

Proof. By inspection of the $\textit{exact-class}(T)$ rules in Figure 10. □

We define substitution for class declarations to perform the substitution only on the `extends` clause of nested classes; substitution of a base value b for variable x does not affect the fields and methods of a class; substitution within fields and methods is performed in the static semantics (see Figure 14).

Definition A.32

$$(C\ \texttt{ext}\ T\ \{\overline{L}\ \vec{F}\ \overline{M}\})\{b/x\} = C\ \texttt{ext}\ T\{b/x\}\ \{\overline{L}\{b/x\}\ \vec{F}\ \overline{M}\}$$

The rules for $CT(\Gamma, T_0, T)$ and $\Gamma \vdash p\ \texttt{final}\ T$ are mutually recursive. We prove a single substitution lemma to cover both judgments. Because there is no subsumption rule for the judgment $\Gamma \vdash p\ \texttt{final}\ T$, it is not closed under substitution, but we can state a weaker property.

Lemma A.33 (Path and *CT* substitution) If $x : T_x \in \Gamma$ and $\vdash b : T_x$, then

(i) if $\Gamma \vdash p\ \texttt{final}\ T$ then $\Gamma\{b/x\} \vdash p\{b/x\}\ \texttt{final}\ T_0$, where $\Gamma\{b/x\} \vdash T_0 \leq T\{b/x\}$. Moreover, if T is a simple class, then T_0 is a simple class; and

(ii) if $CT(\Gamma, T_2, T_1) = C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}$ then $CT(\Gamma\{b/x\}, T_2\{b/x\}, T_1\{b/x\}) = C'\ \texttt{ext}\ T_s'\ \{\overline{L}'\ \vec{F}\ \overline{M}\}$ for some $\overline{L}'$, T_s' and C' where $\overline{L}\{b/x\} \Subset \overline{L}'$, and $\Gamma\{b/x\} \vdash T_s' \leq T_s\{b/x\}$

Proof. Assume $x : T_x \in \Gamma$ and $\vdash b : T_x$. The proof is by induction on derivations of the form $\Gamma \vdash p\ \texttt{final}\ T$ and $CT(\Gamma, T_2, T_1) = C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\}$.

Case F-LOC, F-NULL:

Either $p = \ell_P$ or $p = \texttt{null}$, and so $p\{b/x\} = p$. Both (i) and (ii) hold trivially.

Case F-VAR:

If $p = y$ where $y \neq x$, then $p\{b/x\} = p = y$. By F-VAR, we have $y : T \in \Gamma$, and therefore, $y : T\{b/x\} \in \Gamma\{b/x\}$. Thus, $\Gamma\{b/x\} \vdash y$ `final` $T\{b/x\}$, proving (i), and (ii) holds trivially.

If $p = x$, then $T = T_x$ and $p\{b/x\} = b$. Since $\vdash b : T_x$, then $\vdash b$ `final` T_0 for some T_0 where $\vdash T_0 \leq T_x$. This is because if $b =$ `null`, then clearly $\vdash b$ `final` T_x, and if $b = \ell_P$ for some P, then $\vdash P \leq T_x$ and $\vdash b$ `final` P. Note that in either case, if T_x is a simple class, then $\vdash b$ `final` P' for some simple class P'.

Thus, by Lemma A.22, $\Gamma\{b/x\} \vdash b$ `final` T_0 and, by Lemma A.23, $\Gamma\{b/x\} \vdash T_0 \leq T_x$. Since $\vdash b : T_x$, by Lemma A.24 we have $\vdash T_x$ wf, and so it must be the case that x does not occur in $FV(T_x)$, where $FV(T_x)$ are the free variables of T_x. Thus $T_x = T_x\{b/x\}$. Therefore, $\Gamma\{b/x\} \vdash T_0 \leq T_x\{b/x\}$, and so (i) holds; (ii) holds trivially.

Case F-GET:

$p = q.f$ for some path q and field f, and so $\Gamma \vdash q$ `final` T_q for some T_q and $T = T_f\{q/\texttt{this}\}$ for some T_f where $\mathit{ftype}(\Gamma, T_q, f) =$ `final` T_f.

By the induction hypothesis, $\Gamma\{b/x\} \vdash q\{b/x\}$ `final` T_1, for some T_1 such that $\Gamma\{b/x\} \vdash T_1 \leq T_q\{b/x\}$.

Also, since $\mathit{ftype}(\Gamma, T_q, f) =$ `final` T_f, it must be the case that for some type $T_d \in \mathit{ord}(\Gamma, T_q)$, the judgment $CT(\Gamma, T_q, T_d) = C_d$ `ext` T_{sd} $\{\overline{L_d}\ \vec{F_d}\ \overline{M_d}\}$ occurs in the derivation of $\mathit{ftype}(\Gamma, T_q, f) =$ `final` T_f, and `final` $T_f\ f = e_f$ is in $\vec{F_d}$. So, by the induction hypothesis, it must be the case that $CT(\Gamma\{b/x\}, T_q\{b/x\}, T_d\{b/x\}) = C_d$ `ext` T'_{sd} $\{\overline{L'_d}\ \vec{F'_d}\ \overline{M'_d}\}$ and `final` $T_f\ f = e_f$ is in $\vec{F'_d}$.

Now since $CT(\Gamma, T_q, T_q)$ appears in the derivation of $\mathit{ftype}(\Gamma, T_q, f)$, the inductive hypothesis applies, and thus $\Gamma \vdash \mathit{super}(\Gamma\{b/x\}, T_q\{b/x\}) \leq (\mathit{super}(\Gamma, T_q))\{b/x\}$. Lemma A.30 applies, and so there is some T'_d such that $T'_d \approx T_d\{b/x\}$ and $T'_d \in \mathit{ord}(\Gamma\{b/x\}, T_q\{b/x\})$, and by Lemma A.7, $\mathit{fields}(\Gamma, T'_d, T'_d) = \mathit{fields}(\Gamma, T_d, T_d)$, and so $\mathit{ftype}(\Gamma\{b/x\}, T_q\{b/x\}, f) =$ `final` T_f. Lemma A.11 ensures that $\mathit{ftype}(\Gamma\{b/x\}, T_1, f) =$ `final` T_f.

We can use F-GET to derive $\Gamma\{b/x\} \vdash q\{b/x\}.f$ `final` $T_f\{q\{b/x\}/\texttt{this}\}$, or equivalently, $\Gamma\{b/x\} \vdash q.f\{b/x\}$ `final` $T\{b/x\}$. Note that if T is a simple class then so is $T\{b/x\}$. Thus, (i) holds; (ii) holds trivially.

Case F-RUNTIME:

We have $\Gamma \vdash p$ `final` T' for some T' such that $T \approx T'$. By the induction hypothesis, we have $\Gamma\{b/x\} \vdash p\{b/x\}$ `final` T_0, where $\Gamma\{b/x\} \vdash T_0 \leq T'\{b/x\}$, and so $\Gamma\{b/x\} \vdash T_0 \leq T\{b/x\}$, by $\leq$-RUNTIME and Lemma A.31.

Note that if T cannot be a simple class, as simple classes are not in the domain of *exact-class*. Thus, (i) holds; (ii) holds trivially.

Case CT-OUTER:

$T_1 = T_1\{b/x\} = C$. Also, $TCT(C) = C$ `ext` P $\{\overline{L_t}\ \vec{F}\ \overline{M}\}$ and $\mathit{classes}(\Gamma, T_2, P) = \overline{L_p}$ for some P, $\overline{L_t}$, and $\overline{L_p}$. This means that $\overline{L} = \overline{L_p} \bullet \overline{L_t}\{T_2/\texttt{This}\}$, and $T_s = P$.

By the induction hypothesis, we have $\mathit{classes}(\Gamma\{b/x\}, T_2\{b/x\}, P) = \overline{L}''$ for some $\overline{L}''$ such that $\overline{L_p}\{b/x\} \Subset \overline{L}''$.

Using CT-OUTER, we can conclude that $CT(\Gamma\{b/x\}, T_2\{b/x\}, C) = C$ `ext` P $\{\overline{L}'' \bullet \overline{L_t}\{T_2/\texttt{This}\}\ \vec{F}\ \overline{M}\}$, and moreover, $\overline{L}\{b/x\} = (\overline{L_p} \bullet \overline{L_t}\{T_2/\texttt{This}\})\{b/x\} = (\overline{L_p}\{b/x\} \bullet \overline{L_t}\{T_2/\texttt{This}\}) \Subset (\overline{L}'' \bullet \overline{L_t}\{T_2/\texttt{This}\})$, by Lemma A.13. Thus (ii) is true; (i) holds trivially.

Case CT-NEST:

$T_1 = T.C$, for some T. Therefore $T_1\{b/x\} = T\{b/x\}.C$, $CT(\Gamma, T, T) = C_t$ `ext` T_t $\{\overline{L_t}\ \vec{F_t}\ \overline{M_t}\}$, $\mathit{classes}(\Gamma, T_2, T_s) = \overline{L_s}$, C `ext` T_s $\{\overline{L_n}\ \vec{F}\ \overline{M}\}$ is a member of $\overline{L_t}$, and $\overline{L} = \overline{L_s} \bullet \overline{L_n}\{T_2/\texttt{This}\}$.

By the induction hypothesis, we have $CT(\Gamma\{b/x\}, T\{b/x\}, T\{b/x\}) = C'_t$ `ext` T'_t $\{\overline{L'_t}\ \vec{F'_t}\ \overline{M'_t}\}$ where $\overline{L_t}\{b/x\} \Subset \overline{L'_t}$. Thus C `ext` $T_s\{b/x\}$ $\{\overline{L_n}\{b/x\}\ \vec{F}\ \overline{M}\}$ is a member of $\overline{L'_t}$.

Also by the induction hypothesis, $\mathit{classes}(\Gamma\{b/x\}, T_2\{b/x\}, T_s\{b/x\}) = \overline{L'_s}$ where $\overline{L_s}\{b/x\} \Subset \overline{L'_s}$.

Using CT-NEST, we can show that $CT(\Gamma\{b/x\}, T_2\{b/x\}, T\{b/x\}.C) = C\ \texttt{ext}\ T_s\{b/x\}\ \{\overline{L'_s} \bullet \overline{L_n}\{b/x\}\{T_2\{b/x\}/\texttt{This}\}\ \vec{F}\ \overline{M}\}$.

Moreover, $\overline{L}\{b/x\} = (\overline{L_s} \bullet \overline{L_n}\{T_2/\texttt{This}\})\{b/x\} \Subset \overline{L'_s} \bullet \overline{L_n}\{b/x\}\{T_2\{b/x\}/\texttt{This}\} = \overline{L'_s} \bullet \overline{L_n}\{T_2/\texttt{This}\}\{b/x\}$, by Lemma A.13. Thus, (ii) holds; (i) holds trivially.

Case CT-RUNTIME:

We have $class(T) = P$, $classes(\Gamma, T_0, P) = \overline{L}$ and $CT(\Gamma, T_2, T_1) = \_\ \texttt{ext}\ P\ \{\overline{L} \bullet \emptyset\}$.

By the induction hypothesis, we have $classes(\Gamma\{b/x\}, T_2\{b/x\}, P) = \overline{L'_p}$ for some $\overline{L'_p}$ such that $\overline{L_p}\{b/x\} \Subset \overline{L'_p}$. Also, by Lemma A.31 we have $class(T_1\{b/x\}) = P$ (and also that $T_1\{b/x\} \in \text{dom}(\textit{exact-class})$).

We can use CT-RUNTIME to derive that $CT(\Gamma\{b/x\}, T_2\{b/x\}, T_1\{b/x\}) = \_\ \texttt{ext}\ P\ \{\overline{L'_p} \bullet \emptyset\}$, and so (ii) holds; (i) is true trivially.

Case CT-DEP:

$T_1 = p.\texttt{class}$, $\Gamma \vdash p\ \texttt{final}\ P$, $T_s = P$, and $classes(\Gamma, T_2, P) = \overline{L_p}$ for $\overline{L_p}$ such that $\overline{L} = \overline{L_p} \bullet \emptyset$.

By the induction hypothesis, we have $classes(\Gamma\{b/x\}, T_2\{b/x\}, P) = \overline{L'_p}$ for some $\overline{L'_p}$ such that $\overline{L_p}\{b/x\} \Subset \overline{L'_p}$.

Also by the induction hypothesis, we have $\Gamma\{b/x\} \vdash p\{b/x\}\ \texttt{final}\ T_0$ for some T_0 such that $\Gamma\{b/x\} \vdash T_0 \leq P$. Let $classes(\Gamma\{b/x\}, T_2\{b/x\}, T_0) = \overline{L_0}$. By Lemma A.14, $\overline{L'_p} \Subset \overline{L_0}$, and thus $\overline{L_p}\{b/x\} \Subset \overline{L_0}$.

If $p\{b/x\}.\texttt{class} \notin \text{dom}(\textit{exact-class})$ then using CT-DEP, we can derive that $CT(\Gamma\{b/x\}, T_2\{b/x\}, p\{b/x\}.\texttt{class}) = \_\ \texttt{ext}\ P\ \{\overline{L_0} \bullet \emptyset\}$. By Lemma A.13, $(\overline{L_p} \bullet \emptyset)\{b/x\} = \overline{L_p}\{b/x\} \bullet \emptyset \Subset \overline{L_0} \bullet \emptyset$, and so (ii) holds; (i) is true trivially.

If $p\{b/x\}.\texttt{class} \in \text{dom}(\textit{exact-class})$ then $p = x$, $b = \ell_{P_b}$, for some P_b such that $\Gamma \vdash P_b \leq P$, and thus by A.29 and A.23, $\Gamma\{b/x\} \vdash P_b \leq P$. Let $classes(\Gamma\{b/x\}, T_2\{b/x\}, P_b) = \overline{L_b}$. By Lemma A.14 we have $\overline{L'_p} \Subset \overline{L_b}$. Using CT-RUNTIME, we can deduce that $CT(\Gamma\{b/x\}, T_2\{b/x\}, \ell_{P_b}.\texttt{class}) = \_\ \texttt{ext}\ P_b\ \{\overline{L_b} \bullet \emptyset\}$. Thus, (ii) holds; (i) is true trivially.

Case CT-PRE:

$T_1 = P\texttt{[}T : P.C\texttt{]}$, and $T_s = P$, for some P, T, and C. Moreover $classes(\Gamma, T_2, P) = \overline{L_p}$ for some $\overline{L_p}$ such that $\overline{L} = \overline{L_p} \bullet \emptyset$.

By the induction hypothesis, we have $classes(\Gamma\{b/x\}, T_2\{b/x\}, P) = \overline{L'_p}$ for some $\overline{L'_p}$ such that $\overline{L_p}\{b/x\} \Subset \overline{L'_p}$.

If $T\{b/x\} \notin \text{dom}(\textit{exact-class})$ then using CT-PRE, we can derive that $CT(\Gamma\{b/x\}, T_2\{b/x\}, P\texttt{[}T\{b/x\} : P.C\texttt{]}) = \_\ \texttt{ext}\ P\ \{\overline{L'_p} \bullet \emptyset\}$. Note that $(\overline{L_p} \bullet \emptyset)\{b/x\} = \overline{L_p}\{b/x\} \bullet \emptyset \Subset \overline{L'_p} \bullet \emptyset$, by Lemma A.13, and so (ii) holds; (i) is true trivially.

If $T\{b/x\} \in \text{dom}(\textit{exact-class})$ then $P\texttt{[}T\{b/x\} : P.C\texttt{]} \in \text{dom}(\textit{exact-class})$. Let $\textit{exact-class}(P\texttt{[}T\{b/x\} : P.C\texttt{]}) = P_b$. By Lemma A.27, $\Gamma \vdash P_b \leq P$ and thus by A.29 and A.23, $\Gamma\{b/x\} \vdash P_b \leq P$. Let $classes(\Gamma\{b/x\}, T_2\{b/x\}, P_b) = \overline{L_b}$. By Lemma A.14 we have $\overline{L'_p} \Subset \overline{L_b}$. Using CT-RUNTIME, we can deduce that $CT(\Gamma\{b/x\}, T_2\{b/x\}, \ell_{P_b}.\texttt{class}) = \_\ \texttt{ext}\ P_b\ \{\overline{L_b} \bullet \emptyset\}$. Thus, (ii) holds; (i) is true trivially.

□

Lemma A.33 has some useful corollaries.

Lemma A.34 (Field substitution) If $x : T_x \in \Gamma$ and $\vdash b : T_x$, and $\textit{ftype}(\Gamma, T, f_i) = [\texttt{final}]\ T_i$, then $\textit{ftype}(\Gamma\{b/x\}, T\{b/x\}, f_i) = [\texttt{final}]\ T_i$.

Proof. Follows immediately from the definition of $\textit{ftype}(\Gamma, T, f_i) = [\texttt{final}]\ T_i$ and Lemma A.33. □

Lemma A.35 (Method type substitution) If $x : T_x \in \Gamma$ and $\vdash b : T_x$, and $\textit{mtype}(\Gamma, T_1, T_2, m) = (\vec{x} : \vec{T}) \to T_r$, then $\textit{mtype}(\Gamma\{b/x\}, T_1\{b/x\}, T_2\{b/x\}, m) = (\vec{x} : \vec{T}) \to T_r$.

Proof. Follows immediately from the definition of $\textit{mtype}(\Gamma, T_1, T_2, m) = (\vec{x} : \vec{T}) \to T_r$ and Lemma A.33. □

From Lemma A.33, we can show that subtyping is closed under substitution.

Lemma A.36 (Substitution in $\leq$) If $x : T_x \in \Gamma$ and $\vdash b : T_x$, and $\Gamma \vdash T_1 \leq T_2$, then $\Gamma\{b/x\} \vdash T_1\{b/x\} \leq T_2\{b/x\}$.

Proof. Assume $x : T_x \in \Gamma$ and $\vdash b : T_x$ and $\Gamma \vdash T_1 \leq T_2$. The proof is by induction on the derivation of $\Gamma \vdash T_1 \leq T_2$. Reflexivity is immediate and transitivity follow from the induction hypothesis.

Case $\leq$-EXTENDS:

Then $T_2 = \mathit{super}(\Gamma, T_1)$ and by the definition of *super*, we have $CT(\Gamma, T_1, T_1) = C\ \texttt{ext}\ T_2\ \{\overline{L}\ \vec{F}\ \overline{M}\}$. By Lemma A.33, $CT(\Gamma\{b/x\}, T_1\{b/x\}, T_1\{b/x\}) = C'\ \texttt{ext}\ T_2'\ \{\overline{L'}\ \vec{F}\ \overline{M}\}$, where $\Gamma\{b/x\} \vdash T_2' \leq T_2\{b/x\}$.

By $\leq$-EXTENDS we have $\Gamma\{b/x\} \vdash T_1\{b/x\} \leq T_2'$, and so $\Gamma\{b/x\} \vdash T_1\{b/x\} \leq T_2\{b/x\}$ by transitivity.

Case $\leq$-NEST:

The case holds by the induction hypothesis.

Case $\leq$-RUNTIME:

Then $T_1 \approx T_2$. By Lemma A.31, $T_1\{b/x\} \approx T_2\{b/x\}$ and we can derive $\Gamma\{b/x\} \vdash T_1\{b/x\} \leq T_2\{b/x\}$.

□

The following lemma states that type well-formedness is closed under substitution of base values into types.

Lemma A.37 (Well-formedness substitution) If $x : T_x \in \Gamma$ and $\vdash b : T_x$, and $\Gamma \vdash T$ wf, then $\Gamma\{b/x\} \vdash T\{b/x\}$ wf.

Proof. The proof is by induction on derivation of $\Gamma \vdash T$ wf.

Case WF-OUTER:

Trivial.

Case WF-NEST:

$T = T'.C$, and $T\{b/x\} = T'\{b/x\}.C$. We also have $C\ \texttt{ext}\ T_s\ \{\overline{L}\ \vec{F}\ \overline{M}\} \in \mathit{classes}(\Gamma, T', T')$.

By the inductive hypothesis, $\Gamma\{b/x\} \vdash T'\{b/x\}$ wf. By Lemma A.33, $\mathit{classes}(\Gamma, T', T')\{b/x\} \Subset \mathit{classes}(\Gamma\{b/x\}, T'\{b/x\}, T'\{b/x\})$, and so there is a declaration for C in $\mathit{classes}(\Gamma\{b/x\}, T'\{b/x\}, T'\{b/x\})$. Thus, $\Gamma\{b/x\} \vdash T'\{b/x\}.C$ wf.

Case WF-DEP:

$T = p.\texttt{class}$, and $\Gamma \vdash p\ \texttt{final}\ P$ for some p and P. By Lemma A.33, $\Gamma\{b/x\} \vdash p\{b/x\}\ \texttt{final}\ P'$ for some P'. Thus, $\Gamma\{b/x\} \vdash T\{b/x\}$ wf.

Case WF-PRE:

$T = P\,[T' : P.C]$ for some P andT'. Moreover, $\Gamma \vdash P.C$ wf, $\Gamma \vdash T'$ wf, $\mathit{is\text{-}exact}(T')$, and $\Gamma \vdash T' \leq P.C$.

By the inductive hypothesis, we have $\Gamma\{b/x\} \vdash P.C$ wf and $\Gamma\{b/x\} \vdash T'\{b/x\}$ wf. Inspection of the definition of $\mathit{is\text{-}exact}()$ shows that $\mathit{is\text{-}exact}(T')$ if and only if $\mathit{is\text{-}exact}(T'\{b/x\})$. By Lemma A.36, $\Gamma\{b/x\} \vdash T'\{b/x\} \leq P.C$. Thus, we can use WF-PRE to deduce that $\Gamma\{b/x\} \vdash P\,[T' : P.C]\,\{b/x\}$ wf.

□

Finally, we can state a substitution lemma for typing judgments.

Lemma A.38 (Substitution) If $x : T_x \in \Gamma$ and $\vdash b : T_x$, and $\Gamma \vdash e : T$, then $\Gamma\{b/x\} \vdash e\{b/x\} : T\{b/x\}$.

Proof. Assume $x : T_x \in \Gamma$ and $\vdash b : T_x$. The proof is by induction on the derivation of $\Gamma \vdash e : T$.

Case $e = \ell_P$, $e = \texttt{null}$, $e = y$, $e = \texttt{final}\ T_1\ x = e_1;\ e_2$:

Trivial, as $e\{b/x\} = e$.

Case $e = x$:

$T = T_x$ and $x\{b/x\} = b$, and $\vdash b : T_x$ by assumption.

Case $e = \texttt{final}\ T_1\ y = e_1;\ e_2$:

Follows from induction hypothesis and Lemma A.37.

Case $e = p.f_i$:

Follows from induction hypothesis, Lemma A.33, and Lemma A.34.

Case $e = p.f_i =_{[\texttt{final}]} e_1;\ e_2$:

Follows from induction hypothesis, Lemma A.33, and Lemma A.34.

Case $e = p.m(\vec{v})$:

Follows from induction hypothesis, Lemma A.33, and Lemma A.35.

Case $e = v.\texttt{super}_P.m(\vec{v})$:

Follows from induction hypothesis, Lemma A.33, Lemma A.37, and Lemma A.35.

Case $e = \texttt{new}\ T\ \texttt{as}\ x\ \{\vec{f} = \vec{e}\}$:

Follows from induction hypothesis, and Lemma A.34.

Case T-DEP:

Follows from Lemma A.33.

Case T-$\leq$:

Follows from the induction hypothesis, Lemma A.36, and Lemma A.19.

□

A.7 Subject reduction

Because expressions in our language are evaluated in a heap, to state the subject reduction lemma, we first define a well-typedness condition for heaps and for configurations $\langle H, e\rangle$.

Definition 6.1 (Well-typed heaps) A heap H is *well-typed* if for any memory location $\ell_P \in \text{dom}(H)$,

- $H(\ell_P) = P\ \{\overline{f} = \overline{\ell_{P'}}\}$,
- $\vdash \mathit{ftype}(\emptyset, P, \overline{f}) = \overline{T}$,
- $\vdash \overline{\ell_{P'}} : \overline{T}\{\ell_P/\texttt{this}\}$, and
- $\overline{\ell_{P'}} \subseteq \text{dom}(H)$

Definition 6.2 (Well-formed configurations) A configuration $\langle H, e\rangle$ is *well-formed* if H is well-typed and for any location ℓ_P free in e, $\ell_P \in \text{dom}(H)$.

We state one more lemma before proving subject reduction.

Lemma A.39 (Evaluation contexts) Assume $\Gamma \vdash e : T'$ and $\Gamma \vdash e' : T'$. If $\Gamma \vdash E[e] : T$, then $\Gamma \vdash E[e'] : T$.

Proof. By structural induction on E. □

The subject reduction lemma states that a step taken in the evaluation of a well-formed configuration results in a well-formed configuration.

Lemma 6.3 (Subject reduction) Suppose $\vdash e : T$, $\langle H, e\rangle$ is well-formed, and $\langle H, e\rangle \longrightarrow \langle H', e'\rangle$. Then $\vdash e' : T$ and $\langle H', e'\rangle$ is well-formed.

Proof. The proof is by induction on the derivation of $\langle H, e\rangle \longrightarrow \langle H', e'\rangle$.

Case R-LET:

Then $e = \texttt{final}\ T'\ x = b;\ e$ and $e' = e\{b/x\}$.

If $\vdash \texttt{final}\ T'\ x = b;\ e : T$, then by T-LET, we have $\vdash b : T'$ and $x : T' \vdash e : T$. By Lemma A.38, $\vdash e\{b/x\} : T\{b/x\}$. Since $\vdash T$ wf, we have $FV(T) = \emptyset$, and hence $T\{b/x\} = T$. Thus, $\vdash e\{b/x\} : T$.

Case R-GET:

Then $e = \ell_P.f_i$ and $e' = b_i$ and $T = T_i\{\ell_P/\texttt{this}\}$.

Since $\vdash \ell_P.f_i : T_i\{\ell_P/\texttt{this}\}$, by T-GET, $\vdash \ell_P : T'$ for some T' where $\mathit{ftype}(\emptyset, T', f_i) = [\texttt{final}]\ T_i$.

Since H is well-typed, $H(\ell_P) = \{\overline{f} = \overline{b}\}$, and in particular, $\vdash b_i : T_i\{\ell_P/\texttt{this}\}$, or equivalently $\vdash b_i : T$.

Case R-SET:

Then $e = \ell_P.f_i = b;\ e'$.

Since $\vdash \ell_P.f_i = b;\ e' : T$, then by T-SET, we have: $\vdash \ell_P : T'$ for some T' where $\mathit{ftype}(\emptyset, T', f_i) = [\texttt{final}]\ T_i$, and $\vdash e' : T$, and $\vdash b : T_i\{\ell_P/\texttt{this}\}$.

Now, since H is well-typed and since $H(\ell_P) = P\ \{\overline{f} = \overline{b}\}$, we have: $\vdash \overline{b} : \overline{T}\{\ell_P/\texttt{this}\}$ and $b_i = \texttt{null}$ or $b_i \in \mathrm{dom}(H)$. Now, $H'(\ell_P) = P\ \{\overline{f} = \overline{b'}\}$ where $b'_j = b_j$ for $j = 1, \ldots, i-1, i+1, \ldots, n$ and $b'_i = b$. If $b \neq \texttt{null}$, b is free in e', and therefore, $b \in \mathrm{dom}(H)$ and thus is in $\mathrm{dom}(H')$. Thus H' is well-typed and $\langle H', e' \rangle$ is well-formed.

Case R-CALL:

Then $e = \ell_P.m(\vec{b})$ and $e' = e''\{\ell_P/\texttt{this}, \vec{b}/\vec{x}\}$ where $\mathit{mbody}(P, P, m) = (\vec{x}, e'')$. And $T = T''\{\ell_P/\texttt{this}, \vec{b}/\vec{x}\}$.

Since $\vdash \ell_P.m(\vec{b}) : T''\{\ell_P/\texttt{this}, \vec{b}/\vec{x}\}$, we have by R-CALL, $\mathit{mtype}(\emptyset, P, P, m) = (\vec{x} : \vec{T}) \to T''$.

Now, since $\mathit{mbody}(P, P, m) = (\vec{x}, e'')$ and $\mathit{mtype}(\emptyset, P, P, m) = (\vec{x} : \vec{T}) \to T''$, by Lemma A.2, there is a P' where $\vdash P \leq P'$ such that $\texttt{this} : P', \vec{x} : \vec{T} \vdash e'' : T''$.

By Lemma A.38, we have $\vdash e''\{\ell_P/\texttt{this}, \vec{b}/\vec{x}\} : T''\{\ell_P/\texttt{this}, \vec{b}/\vec{x}\}$, or equivalently $\vdash e' : T$.

Case R-SUPER:

Then $e = \ell_P.\texttt{super}_Q.m(\vec{b})$ and $e' = e''\{\ell_P/\texttt{this}, \vec{b}/\vec{x}\}$, where $\mathit{mbody}(P, Q, m) = (\vec{x}, e'')$. And $T = T''\{\ell_P/\texttt{this}, \vec{b}/\vec{x}\}$.

The proof is similar to previous case, but uses the observation that if $(\emptyset, P, Q) = Q'$, then Q' follows Q in $\mathit{ord}(\emptyset, P)$.

Case R-NEW:

Then $e = \texttt{new}\ T\ \texttt{as}\ x\ \{\vec{f} = \vec{e}\}$ and $e' = \ell_P.\vec{f}' =_{\texttt{final}} \vec{e'};\ \ell_P$, where $\mathit{runtime\text{-}class}(T) = P$. and e'_i is defined as follows: If $f'_i \in \overline{f}$, then $e'_i = e_i\{\ell_P/x\}$; if $f'_i \in \overline{f'} - \overline{f}$, then $e'_i = \mathit{finit}(\emptyset, P, f_i)\{\ell_P/\texttt{this}\}$ where $\overline{f'} = \mathit{fnames}(\emptyset, P)$.

Since $\vdash \texttt{new}\ T\ \texttt{as}\ x\ \{\vec{f} = \vec{e}\} : T$ by T-NEW, we have $\mathit{ftype}(\emptyset, P, \overline{f}) = [\texttt{final}]\ \overline{T}$ and $x : T \vdash \overline{e} : \overline{T}\{x/\texttt{this}\}$ for some $\overline{T}$.

Since $\mathit{runtime\text{-}class}(T) = P$, by T-LOC, we have $\vdash \ell_P : T$.

Thus, to prove that the sequence of field assignments ending in ℓ_P is well-typed and has type T, we need only show for each assignment $\ell_P.f'_i = e'_i$, if $\mathit{ftype}(\emptyset, P, f'_i) = [\texttt{final}]\ T'_i$, then $\vdash e'_i : T'_i\{\ell_P/\texttt{this}\}$.

There are two cases.

1. If $f'_i \in \overline{f}$, then $e'_i = e_i\{\ell_P/x\}$. Since $\mathit{ftype}(\emptyset, P, f'_i) = [\texttt{final}]\ T'_i$, we have by T-NEW $\Gamma, x : P \vdash e_i : T'_i\{x/\texttt{this}\}$. By Lemma A.38, we can derive $\vdash e_i\{\ell_P/x\} : T'_i\{x/\texttt{this}\}\{\ell_P/x\}$ and hence $\vdash e'_i : T'_i\{\ell_P/\texttt{this}\}$.
2. If $f'_i \in \overline{f'} - \overline{f}$, then $e'_i = e''_i\{\ell_P/\texttt{this}\}$, where $\mathit{finit}(\emptyset, P, f'_i) = e''_i$. Since $\mathit{ftype}(\emptyset, P, f'_i) = [\texttt{final}]\ T'_i$, by Lemma A.3, there is a P' such that $\vdash P \leq P'$ and $\texttt{this} : P' \vdash e''_i : T'_i$. Thus, by applying Lemma A.38, we have $\vdash e'_i : T'_i\{\ell_P/\texttt{this}\}$.

Thus, we can derive by T-NEW, $\vdash \ell_P.\vec{f}' =_{\texttt{final}} \vec{e'};\ \ell_P : P$.

Next, we have to show that $\langle H, e' \rangle$ is well-formed. First, observe that ℓ_P is free in e' and is also in $\mathrm{dom}(H')$. Second, $H'(\ell_P) = P\ \{\overline{f'} = \texttt{null}\}$. Since $\vdash \texttt{null} : T$ for any well-formed T, and $\mathit{fnames}(P) = \overline{f'}$, H' is well-typed. Since e' is also well-typed, $\langle H', e' \rangle$ is well-formed.

Case R-CONG:

Then $e = E[e_1]$ and $e' = E[e_1']$, where $\langle H, e_1\rangle \longrightarrow \langle H, e_1'\rangle$.

Since $\vdash E[e_1] : T$, there is a T_1 such that $\vdash e_1 : T_1$.

Thus, by the induction hypothesis, $\vdash e_1' : T_1$ and H' is well-formed.

Finally, by Lemma A.39, $\vdash E[e_1'] : T$.

Case R-NULL:

Then $e = E[N]$ and $e' = \texttt{null}$. If $\Gamma \vdash e : T$, then $\Gamma \vdash \texttt{null} : T$ by T-NULL.

□

A.8 Progress

To prove the progress lemma, we need the following additional lemma.

Lemma A.40 If $\vdash T$ wf then either

(i) $T = TE[p]$, for some access path $p \neq \ell_P$; or

(ii) there is a P such that *runtime-class*$(T) = P$.

Proof. By structural induction on T.

Case $T = C$:

Case (ii) holds.

Case $T = T'.C$:

By the inductive hypothesis, either (i) $T' = TE[p]$, for some access path $p \neq \ell_P$, in which case $T = TE[p]$, using the type evaluation context $TE.C$; or (ii) there is a P' such that *runtime-class*$(T') = P'$, and so *runtime-class*$(T) = P'.C$.

Case $T = p.\texttt{class}$:

If $p \neq \ell_P$ then $T = TE[p]$, and so (i) holds. Otherwise, $p = \ell_P$, and *runtime-classT* $= P$, and so (ii) holds.

Case $T = P\,\texttt{[}T' : P.C\texttt{]}$:

By the inductive hypothesis, either (i) $T' = TE[p]$, for some access path $p \neq \ell_P$, in which case $T = TE[p]$, using the type evaluation context $P\,\texttt{[}TE : P.C\texttt{]}$; or (ii) there is a P' such that *runtime-class*$(T') = P'$, and so *runtime-class*$(T) =$ *prefix*$(P,$ *runtime-class*$(T'),$ *runtime-class*$(T'), P.C)$.

□

The progress lemma states that for any well-formed configuration $\langle H, e\rangle$, either e is a base value ℓ_P or `null`, or $\langle H, e\rangle$ can make a step according to the operational semantics.

Lemma 6.4 (Progress) If $\vdash e : T$, $\vdash T$ wf, $\langle H, e\rangle$ is well-formed, then either $e = b$ or there is a configuration $\langle H', e'\rangle$ such that $\langle H, e\rangle \longrightarrow \langle H', e'\rangle$.

Proof. The proof is by structural induction on e.

Case $e = \ell_P$ or $e = \texttt{null}$:

Then $e = b$.

Case $e = x$:

Impossible since x is not well-typed in the empty environment.

Case $e = \texttt{final}\ T\ x = e_1;\ e_2$:

Since e is well-typed, by T-LET, $\vdash T$ wf. There are three cases for T.

If $T = TE[\texttt{null}]$, then $\langle H, e\rangle$ can make a step by R-NULL.

If $T = TE[p]$ for some access path $p \neq b$, then $\langle H, e\rangle$ can make a step by R-CONG.

If $T = TE[\ell_{P'}]$, then by Lemma A.40, there is a P such that *runtime-class*$(T) = P$. In this case, if $e_1 \neq b$, then $e = E[e_1]$ and $\langle H, e\rangle$ can make a step by R-CONG. Otherwise, e' can make a step by R-LET.

Case $e = p.f_i$:

Since p is well-typed in the empty environment, there are three cases for p.

If $p = \texttt{null}$, then $e = N$ and $\langle H, e\rangle$ can make a step by R-NULL.

If $p = p'.f'$ for some p', then $e = E[p']$ and $\langle H, e\rangle$ can make a step by R-CONG.

If $p = \ell_P$, then by T-GET, we have $\vdash \ell_P : P$. Since H is well-typed, $H(\ell_P) = P\ \{\overline{f} = \overline{b}\}$ and $f_i \in \overline{f}$. Thus, e can make a step by R-GET.

Case $e = p.f_i =_{[\texttt{final}]} e_1;\ e_2$:

Since p is well-typed in the empty environment, there are three cases for p.

If $p = \texttt{null}$, then $e = N$ and $\langle H, e\rangle$ can make a step by R-NULL.

If $p = p'.f'$ for some p', then $e = E[p']$ and $\langle H, e\rangle$ can make a step by R-CONG.

If $p = \ell_P$, then by T-SET, $\vdash \ell_P : P$. If $e_1 = b$, then since H is well-typed, $H(\ell_P) = P\ \{\overline{f} = \overline{b}\}$ and $f_i \in \overline{f}$. In this case e can make a step by R-SET. If $e_1 \neq b$, then $e = E[e_1]$ and $\langle H, e\rangle$ can make a step by R-CONG.

Case $e = p.m(\vec{v})$:

Since p is well-typed in the empty environment, there are three cases for p.

If $p = \texttt{null}$, then $e = N$ and $\langle H, e\rangle$ can make a step by R-NULL.

If $p = p'.f'$ for some p', then $e = E[p']$ and $\langle H, e\rangle$ can make a step by R-CONG.

If $p = \ell_P$, then by T-CALL, $\vdash \ell_P : P$ and $\mathit{mtype}(\emptyset, P, P, m) = (\vec{x} : \vec{T}) \to T'$ for some $\vec{x}$, $\vec{T}$, and T' such that $\#(\vec{x}) = \#(\vec{v})$. Since $\vdash \vec{v} : \vec{T}\{\ell_P/\texttt{this}\}$, $\vec{v} = \vec{b}$. It is easy to see that $\mathit{mbody}(P, P, m) = (\vec{x}, e_0)$. Thus, $\langle H, e\rangle$ can take a step by R-CALL.

Case $e = v.\texttt{super}_Q.m(\vec{v})$:

Since v and $\vec{v}$ are well-typed in the empty environment, $v = b$ and $\vec{v} = \vec{b}$.

If $v = \texttt{null}$, then $e = N$ and $\langle H, e\rangle$ can make a step by R-NULL.

Otherwise, $v = \ell_P$. By T-SUPER, we have $\mathit{mtype}(\emptyset, Q, \mathit{super}(\emptyset, Q), m) = (\vec{x} : \vec{T}) \to T'$ for some $\vec{x}$, $\vec{T}$, and T'. Since $\mathit{super}(\emptyset, Q)$ follows Q in $\mathit{ord}(\emptyset, Q)$, $(\emptyset, P, Q)$ must implement m with the same signature and thus, $\mathit{mtype}(\emptyset, P, Q, m) = (\vec{x} : \vec{T}) \to T'$, where $\#(\vec{x}) = \#(\vec{b})$. It is easy to see that if $(\emptyset, P, Q) = Q'$, $\mathit{mbody}(Q', Q', m) = (\vec{x}, e_0)$. In this case e' can make a step by R-SUPER.

Case $e = \texttt{new}\ T\ \texttt{as}\ x\ \{\overline{f} = \vec{e}\}$:

Since $\vdash T$ wf, there are three cases.

If $T = TE[p]$ for some access path p that is not a base value b, then $\langle H, e\rangle$ can make a step by R-CONG.

If $T = TE[\texttt{null}]$, then $\langle H, e\rangle$ can make a step by R-NULL.

If $T = TE[\ell_{P'}]$, then by Lemma A.40, there is a P such that *runtime-class*$(T) = P$. By T-NEW, we have $\mathit{ftype}(\emptyset, P, \overline{f}) = [\texttt{final}]\ \overline{T}$, and it is easy to see that $\overline{f} \subseteq \mathit{fnames}(\emptyset, P)$. In this case, $\langle H, e\rangle$ can make a step by R-NEW.

□

A.9 Soundness

Finally, we define the normal form of a configuration, define well-formedness for programs, and state the soundness theorem.

Definition 6.5 (Normal forms) A configuration $\langle H, e\rangle$ is in *normal form* if there is no $\langle H', e'\rangle$ such that $\langle H, e\rangle \longrightarrow \langle H', e'\rangle$.

Definition 6.6 A program $Pr = \langle TCT, e\rangle$ is *well-formed* if $\vdash TCT$ ok and $\emptyset \vdash e : T$ for some T such that $\emptyset \vdash T$ wf.

Theorem 6.7 (Soundness) Given a well-formed program $Pr = \langle TCT, e\rangle$, if the configuration $\langle \emptyset, e\rangle$ is well-formed and $\vdash e : T$, and if $\langle H', e'\rangle$ is a normal form such that $\langle \emptyset, e\rangle \longrightarrow^* \langle H', e'\rangle$, then e' is either a location $\ell_P \in \mathrm{dom}(H')$ or `null` and $\vdash e' : T$.

Proof. Immediate from Lemma 6.3 and Lemma 6.4. □