Nested Intersection for Scalable Software Composition

(Technical Report)

Nathaniel Nystrom   Xin Qi   Andrew C. Myers
Computer Science Department
Cornell University
{nystrom,qixin,andru}@cs.cornell.edu

Abstract

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

1. Introduction

Most software is constructed by extending and composing existing code. Existing mechanisms like class inheritance address the problem of code reuse and extension for small or simple extensions, but do not work well for larger bodies of code such as compilers or operating systems, which contain many mutually dependent classes, functions, and types. Moreover, these mechanisms do not adequately support *composition* of multiple interacting classes. Better language support is needed.

This paper introduces the language J& (pronounced "Jet"), which supports the scalable, modular composition and extension of large software frameworks. J& builds on the Java-based language Jx, which supports scalable extension of software frameworks through *nested inheritance* [35]. J& adds a new language feature, *nested intersection*, which enables composition of multiple software frameworks to obtain a software system that combines their functionality.

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

A motivating example for software composition is the problem of combining domain-specific compiler extensions. We demonstrate the utility of nested intersection through a J& compiler framework for implementing domain-specific extensions to the Java language. Using the framework, which is based on the Polyglot compiler framework [36], one can choose useful language features for a given application domain from a "menu" of available options, then compose the corresponding compilers to obtain a compiler for the desired language.

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

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

The first three of these requirements correspond to Wadler's *expression problem* [49]. Scalability (4) is often but not necessarily satisfied by supporting separate compilation; it is important for extending large software. Non-destructive extension (5) enables existing clients of the base system and also the extended system itself to interoperate with code and data of the base system, an important requirement for backward compatibility. Nested inheritance [35] addresses the first five requirements, but it does not support extension composition. Nested intersection adds this capability.

This paper describes nested intersection in the J& language and our experience using it to compose software. Section 2 considers a particularly difficult instantiation of the problem of scalable extensibility and composition—the extension and composition of compilers—and gives an informal introduction to nested intersection and J&. Nested intersection creates several interesting technical challenges, such as the problem of resolving conflicts among composed packages; this topic and a detailed discussion of language semantics are presented in Section 3. Section 4 then de-

This technical report expands on the paper of the same name submitted to OOPSLA 2006.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.

scribes how nested intersection is used to extend and compose compilers. The implementation of J& is described in Section 5, and Section 6 describes experience using J& to implement and compose extensions in the Polyglot compiler framework and in the Pastry framework for building peer-to-peer systems [44]. Related work is discussed in Section 7, and the paper concludes in Section 8. The appendix gives a formal operational semantics and a type system for J&.

2. Nested intersection

Nested intersection supports scalable extension of a base system and scalable composition of those extensions. Consider building a compiler with composable extensions. A compiler is of course not the only system for which extensibility is useful; other examples include user interface toolkits, operating systems, game engines, web browsers, and peer-to-peer networks. However, compilers are a particularly challenging domain because a compiler has several different interacting dimensions along which it can be extended: syntax, types, analyses, and optimizations.

2.1 Nested inheritance

Nested intersection builds on previous work on nested inheritance [35]. Figure 1(a) shows a fragment of J& code for a simple compiler for the lambda calculus extended with pair expressions. This compiler translates the lambda calculus with pairs into the lambda calculus without pairs.

Nested inheritance is inheritance of *namespaces*: packages and classes. In J&, packages are treated like classes with no fields, methods, or constructors. A namespace may contain other namespaces. A namespace may also extend another namespace, inheriting all its members, including nested namespaces. As with ordinary inheritance, the meaning of code inherited from the base namespace is as if it were copied down from the base. A derived namespace may *override* any of the members it inherits, including nested classes and packages.

As with virtual classes [29, 30, 19], overriding of a nested class does not replace the original class, but instead refines, or *further binds* [29], it. If a namespace T' extends another namespace T that contains a nested namespace $T.C$, then $T'.C$ inherits members from $T.C$ as well as from $T'.C$'s explicitly named base namespaces (if any). Further binding thus provides a limited form of multiple inheritance: *explicit inheritance* from the named base of $T'.C$ and *induced inheritance* from the original namespace $T.C$. Unlike with virtual classes, $T'.C$ is also a subtype of $T.C$. In Figure 1(a), the `pair` package extends the `base` package, further binding the `Visitor`, `TypeChecker`, and `Compiler` classes, as illustrated by the `base` and `pair` boxes in the inheritance hierarchy of Figure 2. The class `pair.TypeChecker` is a subclass of both `base.TypeChecker` and `pair.Visitor` and contains both the `visitAbs` and `visitPair` methods.

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

In Figure 1(a), the name `Visitor`, in the context of the `base` package, refers to `base.Visitor`. In the context of `pair`, which inherits from `base`, `Visitor` refers to `pair.Visitor`. Thus, when the method `accept` is called on an instance of `pair.Pair`, it must be called with a `pair.Visitor`, *not* with a `base.Visitor`. This allows `Pair`'s `accept` to invoke the `visitPair` method of the parameter *v*.

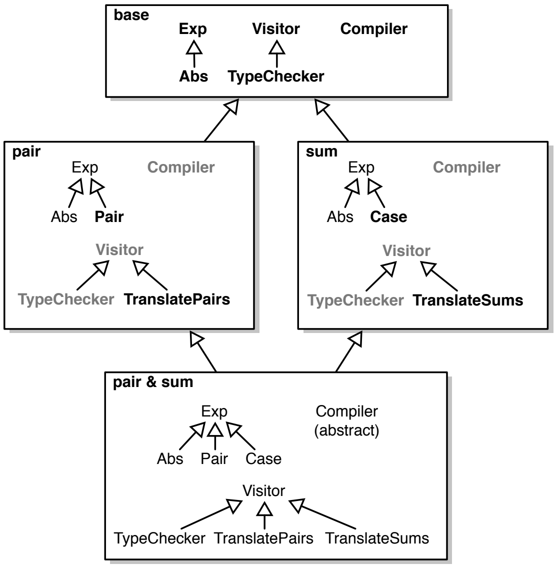

Figure 2. Inheritance hierarchy for compiler composition

Late binding applies to supertype declarations as well. Thus, `pair.Emitter` extends `pair.Visitor` and inherits its `visitPair` method. Late binding of supertype declarations thus provides a form of *virtual superclasses* [30, 15], permitting inheritance relationships among the nested namespaces to be preserved when inherited into a new enclosing namespace. The class hierarchy in the original namespace is replicated in the derived namespace, and in that derived namespace, when a class is further bound, new members added into it are automatically inherited by subclasses in the new hierarchy.

Sets of mutually dependent classes may be extended at once. by grouping them into a namespace. For example, the classes `Exp` and `Visitor` in the `base` package are mutually dependent. Ordinary class inheritance does not work because the extended classes need to know about each other: the `pair` compiler could define `Pair` as a new subclass of `Exp`, but references within `Exp` to class `Visitor` would refer to the old `base` version of `Visitor`, not the appropriate one that understands how to visit pairs. With nested inheritance of the containing namespace, late binding of type names ensures that relationships between classes in the original namespace are preserved when these classes are inherited into a new namespace.

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

2.2 Extensibility requirements

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

Orthogonal extension. Compiler frameworks must support the addition of both new data types (e.g., abstract syntax, types, dataflow analysis values) and operations on those types (e.g., type checking, optimization, translation). It is well known that there is a tension between extending types and extending the procedures that manipulate them [42]. Nested inheritance solves this problem

```
package base;

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

```
package pair extends base;

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

(a) Lambda calculus + pairs compilers

```
package sum extends base;

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

(b) Lambda calculus + sums compiler

```
package pair_and_sum extends pair & sum;

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

(c) Conflict resolution

Figure 1. Compiler composition

because late binding of type names causes inherited methods to operate automatically on data types further bound in the inheriting context.

Type safety. Nested inheritance is also type-safe [35]. Dependent classes ensure that extension code cannot use objects of the base system or of other extensions as if they belonged to the extension, which could cause run-time errors.

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

Non-destructive extension. Nested inheritance does not affect the base code, so it is a non-destructive extension mechanism, unlike open classes [12] and aspects [27]. Therefore, base code and extended code can be used together in the same system, which is important in extensible compilers because the base language is often used as a target language in an extended compiler.

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

2.3 Composition

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

For two namespaces S and T, $S\,\&\,T$ is the *intersection* of these two namespaces. Nested intersection is a form of multiple inheritance implemented using *intersection types* [43, 13]: $S\,\&\,T$ inherits from and is a subtype of both S and T.

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

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

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

3. Semantics of J&

This section gives an overview of the static and dynamic semantics of J&. A formal presentation of the J& type system is omitted for space but can be found in an associated technical report [37].

3.1 Dependent classes and prefix types

In most cases, J& code looks and behaves like Java code. However, unqualified type names are really syntactic sugar for nested classes of dependent classes and prefix types, introduced in Jx [35].

The *dependent class* *p*`.class` represents the run-time class of the object referred to by the *final access path p*. A final access path is either a final local variable, including `this` and final formal parameters, or a field access *p*`.f`, where *p* is a final access path and `f` is a final field of *p*. In general, the class represented by *p*`.class` is statically unknown, but fixed: for a particular *p*, all instances of *p*`.class` have the same run-time class, and not a proper subclass, as the object referred to by *p*.

The *prefix type* *P*`[`*T*`]` represents the enclosing namespace of the class or interface *T* that is a subtype of the namespace *P*. It is required that *P* be a non-dependent type: either a top-level namespace *C* or a namespace of the form $P'.C$. In typical use *T* is a dependent class. *P* may be either a package or a class. Prefix types provide an unambiguous way to name enclosing classes and packages of a class without the overhead of storing references to enclosing instances in each object, as is done in virtual classes. Indeed, if the enclosing namespace is a package, there are no run-time instances of the package that could be used for this purpose.

Late binding of types is provided by interpreting unqualified names as members of the dependent class `this.class` or of a prefix type of `this.class`. The compiler resolves the name *C* to the type `this.class`.*C* if the immediately enclosing class contains or inherits a nested namespace named *C*. Similarly, if an enclosing namespace *P* other than the immediately enclosing class contains or inherits *C*, the name *C* resolves to *P*`[this.class]`.*C*. Derived namespaces of the enclosing namespace may further bind and refine *C*. The version of *C* selected is determined by the run-time class of `this`.

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

Both dependent classes and prefixes of dependent classes are *exact types* [5]: all instances of these types have the same run-time class, but that class is statically unknown in general. Simple types like `base.Visitor` are not exact since variables of this type may contain instances of any subtype of `Visitor`.

J& provides a form of *family polymorphism* [17]. All types indexed by a given dependent class—the dependent class itself, its prefix types, and its nested classes—are members of a *family* of interacting classes and packages. By initializing a variable with instances of different classes, the same code can refer to classes in different families with different behaviors. In the context of a given class, other classes and packages named using `this.class` are in the same family as the actual run-time class of `this`. In Figure 1(a), `pair.Pair.accept`'s formal parameter `v` has type `base[this.class]`.`Visitor`. If `this` is a `pair.Pair`, `base[this.class]`.`Visitor` must be a `pair.Visitor`, ensuring the call to `visitPair` is permitted.

The type system ensures that types in different families (and hence indexed by different access paths) cannot be confused with each other accidentally: a `base` object cannot be used where a `pair` object is expected, for example. However, casts with run-time type checks allow an escape hatch that can enable wider code reuse. Casting an object to a dependent class *p*`.class` checks that the object has the same run-time class as *p*. This feature allows objects indexed by different access paths to be explicit coerced into another family of types.

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

class A1 extends A {      class A2 extends A {
  class B { }               class B { }
  class C { }               class C { }
  void m() { }              void m() { }
  void p() { }              void p() { }
}                         }

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

Figure 3. Multiple inheritance with name conflicts

Nested inheritance can operate at every level of the containment hierarchy. Unlike with virtual classes [19], in J& a class nested within one namespace can be subclassed by a class in a different namespace. For example, suppose a collections library `util` is implemented in J& as a set of mutually dependent interoperating classes. A user can extend the class `util.LinkedList` to a class `MyList` not nested within `util`. A consequence of this feature is that a prefix type *P*`[`*T*`]` may be defined even if *T* is not directly nested within *P* or within a subtype of *P*. When the current object `this` is a `MyList`, the prefix type `util[this.class]` is well-formed and refers to the `util` package, even though `MyList` is not a member class of `util`.

To ensure soundness, the type *p*`.class` is well-formed only if *p* is final. However, to improve expressiveness and to ease porting of Java programs to J&, a non-final local variable *x* may be *implicitly coerced* to the type *x*`.class` under certain conditions. When *x* is used as an actual argument of a method call, a constructor call, or a `new` expression, or as the source of a field assignment, and if *x* is not assigned in the expression, then it can be implicitly coerced to type *x*`.class`. Consider the following code fragment using the classes of Figure 1(a):

```
base.Exp e = new pair.Pair();
e.accept(new base[e.class].TypeChecker());
```

In the call to `accept`, `e` is never assigned and hence its run-time class does not change between the time `e` is first evaluated and method entry. If `e` had been assigned, say to a `base.Exp`, the `new` expression would have allocated a `base.TypeChecker` and passed it to `pair.Pair.accept`, leading to a run-time type error. Implicit coercion is not performed for field paths, since it would require reasoning about aliasing and is in general unsafe for multithreaded programs.

3.2 Intersection types

Nested intersection of classes and packages in J& is provided in the form of *intersection types* [43, 13]. An intersection type *S* `&` *T* inherits all members of its base namespaces *S* and *T*. With nested intersection, the nested namespaces of *S* and *T* are themselves intersected.

To support composition of classes and packages inherited more than once, J& provides *shared* multiple inheritance: when a subclass (or subpackage) inherits from multiple base classes, the new subclass may inherit the same superclass from more than one immediate superclass; however, instances of the subclass will not contain multiple subobjects for the common superclass. For instance, `pair_and_sum.Visitor` in Figure 1(c) inherits from `base.Visitor` only once, not twice through both `pair` and `sum`. Similarly, the package `pair_and_sum` contains only one `Visitor` class, the composition of `pair.Visitor` and `sum.Visitor`.

3.3 Name conflicts

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

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

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

On the other hand, `A1` and `A2` both declare independent nested classes `C`. Even though these classes have the same name, they are assumed to be unrelated. The class `(A1 & A2).C` is *ambiguous*. In fact, `A1 & A2` contains two nested classes named `C`, one that is a subclass of `A1.C` and one a subclass of `A2.C`. Class `D` and its subclasses can resolve the ambiguity by exploiting prefix type notation: `A1[D].C` refers to the `C` from `A1` and `A2[D].C` refers to the `C` from `A2`. In `A1`, references to the unqualified name `C` are interpreted as `A1[this.class].C`. If `this` is an instance of `D`, these references refer to the `A1.C`. Similarly, references to `C` in `A2` are interpreted as `A2[this.class].C`, and when `this` is a `D`, these references refer to `A2.C`.

A similar situation occurs with the methods `A1.p` and `A2.p`. Again, `D` inherits both versions of `p`. Callers of `D.p` must resolve the ambiguity by up-casting the receiver to specify which one of the methods to invoke. This solution is also used for nonvirtual "`super`" calls. If the superclass is an intersection type, the call may be ambiguous. The ambiguity is resolved by up-casting the special receiver `super` to the desired superclass.

Finally, two or more intersected classes may declare methods that override a method declared in a common base class. In this case, illustrated by the method `m` in Figure 3, the method in the intersection type `A1 & A2` is considered *abstract*. Because it cannot override the abstract method, the intersection is also abstract and cannot be instantiated. Subclasses of the intersection type (`D`, in the example), must override `m` to resolve the conflict, or else also be declared abstract.

3.4 Anonymous intersections

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

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

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

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

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

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

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

Figure 4. Conflicts introduced by late binding

If `A` and `B` have a name conflict that causes their intersection to be an abstract class, a class body must be provided to resolve the conflict.

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

3.5 Prefix types and intersections

Unlike with virtual classes [19], it is possible in J& to extend classes nested within other namespaces. Multiple nested classes or a mix of top-level and nested classes may be extended, resulting in an intersection of several types with different containers. This flexibility is needed for effective code reuse but complicates the definition of prefix types. Consider this example:

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

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

Prefix types ensure the desired type equivalence. Two types P and P' are *related by further binding* if they both contain nested types $P.C$ and $P'.C$ that are inherited from or further bind a common type $P''.C$. We write $P \sim P'$ for the symmetric, transitive closure of this relation. In general, if $P \sim P'$, then $P[T]$ and $P'[T]$ should be equivalent. The prefix type $P[T]$ is defined as the intersection of all types P', where $P \sim P'$ where T has a supertype nested in P and a supertype nested in P'. Using this definition `A`, `A1` and `A2` are all transitively related by further binding. Thus, `A[C]`, `A1[C]`, and `A2[C]` are all equivalent to `A1 & A2`.

Prefix types impose some restrictions on which types may be intersected. If two classes T_1 and T_2 contain conflicting methods,

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

Figure 5. Constructors of a shared superclass

then their intersection is abstract, preventing the intersection from being instantiated. If T_1 or T_2 contain member classes, a prefix type of a dependent class bounded by one of these member classes could resolve to the intersection $T_1 \& T_2$. To prevent these prefix types from being instantiated, all member classes of an abstract intersection are also abstract.

3.6 Constructors

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

When a class declares a `final` field, it must ensure the field is initialized. Since constructors are inherited from base classes that are unaware of the new field, J& requires that if the field declaration does not have an explicit initializer, all inherited constructors must be overridden to initialize the field.

To ensure fields can be initialized to meaningful values, constructors are inherited only via induced inheritance, not via explicit inheritance. That is, the class $T'.C$ inherits constructors from $T.C$ when T is a supertype of T', but not from other superclasses of $T'.C$. If a constructor were inherited from both explicit and induced superclasses, then every class that adds a `final` field would have to override the default `Object()` constructor to initialize the field. Since no values are passed into this constructor, the field may not be able to be initialized meaningfully.

Since a dependent class *p*.`class` may represent any subclass of *p*'s statically known type, a consequence of this restriction is that *p*.`class` can only be explicitly instantiated if *p*'s statically known class is `final`; in this case, since *p*.`class` is guaranteed to be equal to that `final` class, a constructor with the appropriate signature exists. The restriction does not prevent nested classes of dependent classes from being instantiated.

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

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

In the first case, if all calls to the shared superclass's constructor originate from the same call site, which is multiply inherited into the intersection, then every call to the shared constructor will pass it the same arguments. In this case, the programmer need do nothing; the operational semantics of J& will ensure that the shared constructor is invoked exactly once.

For example, in Figure 5, the implicit class `D.C` is a subclass of `B1.C`&`B2.C` and shares the superclass `A`. Since `B1.C` and `B2.C` both inherit their `C(int)` constructor from `B.C`, both inherited constructors invoke the `A` constructor with the same arguments. There is no conflict and the compiler need only ensure that the constructor of `A` is invoked exactly once, before the body of `D.C`'s constructor is executed. Similarly, if the programmer invokes:

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

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

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

In Figure 5, if one or both of `B1` and `B2` were to override the `C(int)` constructor, then `B1.C` and `B2.C` would have different constructors with the same signature. One of them might change how the `C` constructor invokes `A(int)`. To resolve the conflict, `D` must further bind `C` to specify how `C(int)` should invoke the constructor of `A`. This behavior is similar to that of constructors of shared virtual base classes in C++.

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

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

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

3.7 Type substitution

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

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

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

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

For clarity, each occurrence of `this` has been labeled with an abbreviation of its declared type. Since the formal type $\texttt{base[this}_{\texttt{V}}\texttt{.class].Abs}$ depends on the receiver $\texttt{this}_{\texttt{V}}$, the type of the actual argument $\texttt{this}_{\texttt{A}}$ must depend on the receiver *v*.

The type checker substitutes the actual argument types for dependent classes occurring in the formal parameter types. In this example, the receiver `v` has the type $\texttt{base[this}_{\texttt{A}}\texttt{.class].Visitor}$. Substituting this type for $\texttt{this}_{\texttt{V}}\texttt{.class}$ in the formal parameter type $\texttt{base[this}_{\texttt{V}}\texttt{.class].Abs}$ yields $\texttt{base[base[this}_{\texttt{A}}\texttt{.class].Visitor].Abs}$, which is equivalent to $\texttt{base[this}_{\texttt{A}}\texttt{.class].Abs}$.

The type substitution semantics of J& generalize the original Jx substitution rules [35] to increase expressive power. However, to

```
package pair;                package pair_and_sum
                                     extends pair;

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

Figure 6. Static virtual types

ensure soundness, some care must be taken. If the type of v were base.Visitor, then v might refer at run time to a pair.Visitor while at the same time this$_A$ refers to a base.Abs. Substitution of base.Visitor for this$_V$.class in the formal parameter type would yield base[base.Visitor].Abs, which is equivalent to base.Abs. Since the corresponding actual argument has type base[this$_A$.class].Abs, which is a subtype of base.Abs, the call would incorrectly be permitted, leading to a potential run-time type error. The problem is that there is no guarantee that the run-time classes of this$_A$ and v both have the same enclosing base package.

To remedy this problem, type substitution must satisfy the requirement of *exactness preservation*; that is, when substituting into an exact type—a dependent class or a prefix of a dependent class—the resulting type must also be exact. This ensures that the run-time class or package represented by the type remains fixed. Substituting the type base[this$_A$.class].Visitor. for this$_V$.class is permitted since both base[this$_V$.class] and base[this$_A$.class] are exact. However, substituting base.Visitor for this$_V$.class is illegal since base is not exact; therefore, a call to visitAbs where v is declared to be a base.Visitor is not permitted.

Implicit coercion of a non-final local variable *x* to dependent class *x*.class, described in Section 3.1, enhances the expressiveness of J& when checking calls by enabling *x*.class to be substituted for a formal parameter or this. Since this substitution preserves exactness, the substitution is permitted. If *x*'s declared type were substituted for the formal instead, exactness might not have been preserved.

3.8 Static virtual types

Dependent classes and prefix types enable classes nested within a given containment hierarchy of packages to refer to each other without statically binding to a particular fixed package. This allows derived packages to further bind a class while preserving its relationship to other classes in the package. It is often useful to refer to other classes *outside* the class's containment hierarchy without statically binding to a particular fixed package. J& provides *static virtual types* to support this feature. Unlike virtual types in BETA [29], a static virtual type is an attribute of an enclosing package or class rather than of an enclosing object.

In Figure 6, the package pair declares a static virtual type TgtExp representing an expression of the target language of a rewriting pass, in this case an expression from the base compiler. The rewrite method takes an expression with type pair[this.class].Exp and returns a base.Exp. The pair_and_sum package extends the pair package and further binds TgtExp to pair.Exp. A static virtual type can be further bound to any subtype of the original bound. Because pair_and_sum.TgtExp is bound to pair.Exp, the method pair_and_sum.Rewriter.rewrite must return a pair.Exp, rather than a base.Exp as in pair.Rewriter.rewrite.

With intersections, a static virtual type may be inherited from more than one superclass. Consider the declarations in Figure 7. Class B1 & B2 inherits T from both B1 and B2. The type (B1 & B2).T must be a subtype of both A1 and A2; thus, (B1 & B2).T is bound to A1 & A2.

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

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

Figure 7. Static virtual types example

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

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

The exact virtual type B.T is equivalent to the dependent class (new A).class; that is, B.T contains only instances with run-time class A and not any subtype of A. Similarly, B2.T is equivalent to (new A2).class. If a variable b has declared type B, then an instance of b.class.T may be either a A or a A2, depending on the run-time class of b.

3.9 Packages

J& supports inheritance of packages, including multiple inheritance. In fact, the most convenient way to use nested inheritance is usually at the package level, because large software is usually contained inside packages, not classes. The semantics of prefix packages and intersection packages are similar to those of prefix and intersection class types, described above. Since packages do not have run-time instances, the only exact packages are prefixes of a dependent class nested within the package, e.g., pkg[x.class], where x is an instance of class pkg.C.

4. Composing compilers

Using the language features just described we can construct a composable, extensible compiler. In this section, we sketch the design of such a compiler. Most of the design described here was used in our port to J& of the Polyglot compiler framework [36] except where necessary to maintain backward compatibility with the Java version of Polyglot.

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

4.1 Orthogonal extension

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

Because the Visitor design pattern [21] is used to implement compiler passes, when a new AST node class is added to an extension's ast package, a visit method for the class

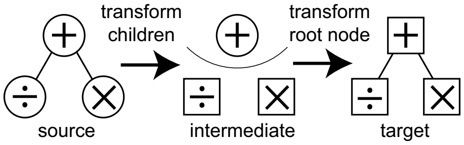

Figure 8. AST transformation

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

4.2 Composition

Independent compiler extensions can be composed using nested intersection with minimal effort. If the two compiler extensions are orthogonal, as for example with the product and sum type compilers of Section 2.3, then composing the extensions is trivial: the `main` method needs to be overridden in the composing extension to specify the order in which passes inherited from the composed extensions should run.

If the language extensions have conflicting semantics, this will often manifest as a name conflict when intersecting the classes within the two compilers. These name conflicts must be resolved to be able to instantiate the composed compiler, forcing the compiler developer to reconcile the conflicting language semantics.

It is undecidable to determine precisely whether two programs, including compilers, have conflicting semantics that prevent their composition. Several conservative algorithms based on program slicing have been proposed for integrating programs [23, 2, 31]. These algorithms detect when two procedures are semantically compatible, or *noninterfering*. Interprocedural program integration [2] requires the whole program and it is unclear whether the algorithm can scale up to large programs. Formal specification offers a way to more precisely determine if two programs have semantic conflicts.

4.3 Extensible rewriters

One challenge for building an extensible compiler is to implement transformations between different program representations. In Figure 1, for example, a compiler pass transforms expressions with pairs into lambda calculus expressions. For a given transformation between two representations, compiler extensions need to be able to scalably and modularly extend both the source and target representations and the transformation itself. However, if the extensions to the source and target representations do not interact with a transformation, it should not be necessary to change the transformation.

Consider an abstract syntax tree (AST) node representing a binary operation. As illustrated in Figure 8, most compiler transformations for this kind of node would recursively transform the two child nodes representing the operands, then invoke pass-specific code to transform the binary operation node itself, in general constructing a new node using the new children. This generic code can be shared by many passes.

However, code for a given base compiler transformation might not be aware of the particular extended AST form used by a given compiler extension. The extension may have added new children to the node in the source representation of which the transformation is unaware. It is therefore hard to write a reusable compiler pass; the pass may fail to transform all the node's children or attributes.

In the `pair` compiler of Figure 1, the `TranslatePairs` pass transforms `pair` AST nodes into `base` AST nodes. If this compiler pass is reused in a compiler in which expressions have, say, additional type annotations, the source and target languages node will have children for these additional annotations, but the pass will not be aware of them and will fail to transform them.

Static virtual types (Section 3.8) are used to make a pass aware of any new children added by extensions of the source language, while preserving modularity. The solution is for the compiler to explicitly represent nodes in the intermediate form as trees with a root in the source language but children in the target language, corresponding to the middle tree of Figure 8. This design is shown in Figure 9. In the example of Figure 1, this can be done by creating, for both the source (i.e., `pair`) and target (i.e., `base`) language, packages `ast_struct` defining just the structure of each AST node. The `ast_struct` packages are then extended to create `ast` packages for the actual AST nodes. Finally, a package is created *inside each visitor class* for the intermediate form nodes of that visitor's specific source and target language.

In the `ast_struct` package, children of each AST node reside in a `child` virtual package. The `ast` package extends the `ast_struct` package and further binds `child` to the `ast` package itself; the node classes in `ast` have children in the same package as their parent.

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

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

5. Implementation

We implemented the J& compiler in Java using the Polyglot framework [36]. The compiler is a 2700-LOC (lines of code, excluding blank and comment lines) extension of the Jx compiler [35], itself a 22-kLOC extension of the Polyglot base Java compiler.

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

5.1 Translating classes

Each explicit J& class is translated into four Java classes: an instance class, a subobject class, a class class, and a method interface. Figure 10 shows a simplified fragment of the translation of the code in Figure 1. Several optimizations discussed below are not shown.

At run time, each instance of a J& class T is represented as an instance of T's instance class, $\mathsf{IC}(T)$. Each explicit class has its own instance class. The instance class of an implicit class or intersection class is the instance class of one of its explicit superclasses.

```
package base.ast_struct;

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

```
package base.ast extends ast_struct;

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

```
package base;

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

Figure 9. Extensible rewriting example

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

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

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

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

The class class provides functions for accessing run-time type information to implement `instanceof` and casts, for constructing instances of the class, and for accessing the class class object of prefix types and member types, including static virtual types. The code generated for expressions that dispatch on a dependent class (a `new x.class()` expression, for example) evaluates the dependent class's access path (i.e., `x`) and uses the method `jetGetClass()` to locate the class class object for the type.

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

Each method has an interface nested in the *method interface* of the J& class that first introduced the method. The class class implements the corresponding interfaces for all methods it declares or overrides. The class class of the J& class that introduces a method `m` also contains a method `m$disp`, responsible for method dispatching. The receiver and method arguments as well as a class

```
package base;

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

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

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

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

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

...
```

Figure 10. Fragment of translation of code in Figure 1

class are passed into the dispatch method. The class class argument is used to implement nonvirtual `super` calls; for virtual calls, `null` is passed in and the receiver's class class is used.

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

Each virtual method call is translated into a call to the dispatch method, which does a lookup to find the class class of the most specific implementation. The class class object is cast to the appropriate method interface and then the method implementation is invoked.

As shown in Figure 10, all references to J& objects are of type `JetInst`. The translation mangles method names to handle overloading. Name mangling is not shown in Figure 10 for readability.

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

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

5.2 Translating packages

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

5.3 Java compatibility

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

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

5.4 Optimizations

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

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

6. Experience

6.1 Polyglot

Following the approach described in Section 4, we ported the Polyglot compiler framework and several Polyglot-based extensions, all written in Java, to J&. The Polyglot base compiler is a 31.9 kLOC program that performs semantic checking on Java source code and outputs equivalent Java source code. Special design patterns make Polyglot highly extensible [35]; more than a dozen research projects have used Polyglot to implement various extensions to Java (e.g., JPred [34], JMatch [28], as well as Jx and J&). For this work we ported six extensions ranging in size from 200 to 3000 LOC.

The extensions are summarized in Table 1. The parsers for the base compiler, extensions, and compositions were generated from CUP [24] or Polyglot parser generator (PPG) [36] grammar files. Because PPG supports only single grammar inheritance, grammars were composed manually, and line counts do not include parser code.

The port of the base compiler was our first attempt to port a large program to J&, and was completed by one of the authors within a few days, excluding time to fix bugs in the J& compiler. Porting of each of the extensions took from one hour to a few days. Much of the porting effort could be automated, with most files requiring only modification of `import` statements, as described below in Section 6.3.

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

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

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

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

6.2 Pastry

We also ported the FreePastry peer-to-peer framework [44] version 1.2 to J& and composed a few Pastry applications. The sizes of the original and ported Pastry extensions are shown in Table 3. Excluding bundled applications, FreePastry is 7.1 kLOC.

Name	Extends Java 1.4 . . .	LOC original	LOC ported	% original
`polyglot`	with nothing	31888	27984	87.8
`param`	with infrastructure for parameterized types	513	540	105.3
`coffer`	with resource management facilities similar to Vault [14]	2965	2642	89.1
`j0`	with pedagogical features	679	436	64.2
`pao`	to treat primitives as objects	415	347	83.6
`carray`	with constant arrays	217	122	56.2
`covarRet`	to allow covariant method return types	228	214	93.9

Table 1. Ported Polyglot extensions

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

Table 2. Polyglot composition results: lines of code

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

Pastry implements a distributed hash table. Beehive and PC-Pastry extend Pastry with caching functionality [41]. PC-Pastry uses a simple passive caching algorithm, where lookups are cached on nodes along the route from the requesting node to a node containing a value for the key. Beehive actively replicates objects throughout the network according to their popularity. We introduced a package `cache` containing functionality in common between Beehive and PC-Pastry; the CorONA RSS feed aggregation service [40] was modified to extend the `cache` package rather than Beehive.

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

The J& code for FreePastry is 7.4 kLOC, 300 lines longer than the original Java code. The additional code consists primarily of interfaces introduced to implement network message dispatching.

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

Table 3. Ported Pastry extensions and compositions

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

6.3 Porting Java to J&

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

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

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

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

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

Path aliasing. The port of Pastry and its extensions made more extensive use of field-dependent classes (e.g., this.thePastryNode.class) than the Polyglot port. Several casts needed to be inserted in the J& code for Pastry to allow a type dependent upon one access path to be coerced to a type dependent upon another path. Often, the two paths refer to the same object, ensuring the cast will always succeed. A simple local alias analysis would eliminate the need for many of these casts.

7. Related work

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

Virtual classes. Nested classes in J& are similar to virtual classes [29, 30, 25, 19]. Virtual classes were originally developed for the language BETA [29, 30], primarily for generic programming rather than for extensibility.

Although virtual classes in BETA are not statically type safe, Ernst's generalized BETA (gbeta) language [15, 16] uses path-dependent types, similar to dependent classes in J&, to ensure static type safety. Type-safe virtual classes using path-dependent types were formalized by Ernst et al. in the *vc* calculus [19].

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

Both J& and gbeta provide virtual superclasses, the ability to late-bind a supertype declaration. When the containing namespace of a set of classes is extended via inheritance, the derived namespace replicates the class hierarchy of the original namespace, forming a *higher-order hierarchy* [18]. Because virtual classes are contained in an object rather than in a class, there is no subtyping relationship between classes in the original hierarchy and further bound classes in the derived hierarchy, as there is in J&.

The gbeta language supports multiple inheritance. As in J&, commonly named virtual classes inherited into a class are themselves composed [16]. However, multiple inheritance is limited to other classes nested within the same enclosing instance.

Virtual classes in gbeta support family polymorphism [17]: two virtual classes enclosed by distinct objects cannot be statically confused. When a containing namespace is extended, family polymorphism ensures the static type safety of the classes in the derived family by preventing it from treating classes belonging to the base family as if they belonged to the extension. In gbeta, each object defines a family of classes: the collection of mutually dependent virtual classes immediately nested within it. Because nested classes in J& are attributes of their enclosing class, rather than an enclosing object, J& supports what Clarke et al. [11] call *class-based family polymorphism*. With virtual classes, all members of the family are named from a single "family object", which must be made accessible throughout the system. Moreover, only nested classes of the family object are part of the family. In contrast, with class-based family polymorphism, each dependent class defines a family of classes nested within and also enclosing. By using prefix types, any instance of a class in the family can be used to name the family, not just a single family object.

Tribe [11] is another language that provides a variant of virtual classes. By treating a final access path p as a type, nested classes in Tribe can be considered attributes of an enclosing class as in Jx and J& or as attributes of an enclosing instance as in BETA and its derivatives. This flexibility allows a further bound class to be a subtype of the class it overrides, like in J& but unlike with virtual classes. Tribe also supports multiple inheritance. However, superclasses of a Tribe class must be nested within the same enclosing class, limiting extensibility. This restriction allows the enclosing type to be named using an owner attribute: T.owner is the enclosing class of T.

Concord [26] also provides a type-safe variant of virtual classes. In Concord, mutually dependent classes are organized into *groups*, which can be extended via inheritance. References to other classes within a group are made using types dependent on the current group, MyGrp, similarly to how prefix types are used in J&. Relative supertype declarations provide functionality similar to virtual superclasses. Groups in Concord cannot be nested, nor can groups be multiply inherited.

Multiple inheritance. J& provides multiple inheritance through nested intersection. Intersection types were introduced by Reynolds in the language Forsythe [43] and were used by Compagnoni and Pierce to model multiple inheritance [13]. Cardelli [9] presents a formal semantics of multiple inheritance.

The distinction between name conflicts among methods introduced in a common base class and among methods introduced independently with possibly different semantics was made as early as 1982 by Borning and Ingalls [3]. Many languages, such as C++ [47] and Self [10], treat all name conflicts as ambiguities to be resolved by the caller. Some languages [32, 4, 45] allow methods to be renamed or aliased.

A *mixin* [4, 20], also known as an *abstract subclass*, is a class parameterized on its superclass. Mixins are able to provide uniform extensions, such as adding new fields or methods, to a large number of classes. Mixins can be simulated using explicit multiple inheritance. J& also provides additional mixin-like functionality through virtual superclasses.

Since mixins are composed linearly, a class may not be able to access a member of a given super-mixin because the member is overridden by another mixin. Explicit multiple inheritance imposes no ordering on composition of superclasses.

Traits [45] are collections of abstract and non-abstract methods that may be composed with state to form classes. Since traits do not have fields, many of the issues introduced by multiple inheritance (for example, whether to duplicate code inherited through more than one base trait) are avoided. The code reuse provided by traits is largely orthogonal to that provided by nested inheritance and could be integrated into J&.

Scala Scala [38] is another language that supports scalable extensibility and family polymorphism through a statically safe virtual type mechanism based on path-dependent types. However, Scala's path-dependent type p.type is a singleton type containing only the value named by access path p; in J&, p.class is not a singleton. For instance, new x.class(...) creates a new object of type x.class distinct from the object referred to by x. This difference gives J& more flexibility, while preserving type soundness. Scala provides virtual types, but not virtual classes. It has no analogue to prefix types, nor does it provide virtual superclasses, limiting the scalability of its extension mechanisms. Scala supports composition using traits. Since traits do not have fields, new state cannot be easily added into an existing class hierarchy.

Self types and matching. Bruce et al. [7, 5] introduce *matching* as an alternative to subtyping, with a *self type*, or `MyType`, representing the type of the method's receiver. The dependent class `this.class` is similar but represents only the class referred to by `this` and not its subclasses. Type systems with `MyType` decouple subtyping and subclassing; in PolyTOIL and LOOM, a subclass *matches* its base class but is not a subtype. With nested inheritance, subclasses are subtypes. Bruce and Vanderwaart [8, 6] propose *type groups* as a means to aggregate and extend mutually dependent classes, similarly to Concord's group construct, but using matching rather than subtyping.

Open classes and expanders. An *open class* [12] is a class to which new methods can be added without needing to edit the class directly, or recompile code that depends on the class. Nested inheritance provides similar functionality through class overriding in an extended container. Nested inheritance provides additional extensibility that open classes do not, such as the "virtual" behavior of constructors, and the ability to extend an existing class with new fields that are automatically inherited by its subclasses.

Similar to open classes, *expanders* [50] are a mechanism for extending existing classes. They address the limitations of open classes by enabling classes to be updated not only with new methods, but also with new fields and superinterfaces. Expanders do not change the behavior of existing clients of extended classes. Existing classes are extended with new state using wrapper objects. One limitation of this approach is that object identity is not preserved, which may cause run-time type checks to return incorrect results.

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

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

Class hierarchy composition. Ossher and Harrison [39] propose an approach in which extensions of a class hierarchy are written in separate sparse extension hierarchies containing only new functionality. Extension hierarchies can be merged and naming conflicts detected. However, semantic incompatibilities between extension hierarchies are not detected. Unlike with nested intersection, hierarchies do not nest and there is no subtyping relationship between classes in different hierarchies.

Tarr et al. [48] define a specification language for composing class hierarchies. Rules specify how to merge "concepts" in the hierarchies. Nested intersection supports composition with a rule analogous to merging concepts by name.

Snelting and Tip [46] present an algorithm for composing class hierarchies and a semantic interference criterion. If the hierarchies are *interference-free*, the composed system preserves the original behavior of classes in the hierarchies. J& reports a conflict if composed class hierarchies have a *static interference*, but makes no effort to detect dynamic interference.

Aspect-oriented programming. Aspect-oriented programming (AOP) [27] is concerned with the management of *aspects*, functionality that cuts across modular boundaries. Nested inheritance provides aspect-like extensibility; an extension of a container may implement functionality that cuts across the class boundaries of the nested classes. Aspects modify existing class hierarchies, whereas nested inheritance creates a new class hierarchy, allowing the new hierarchy to be used alongside the old. Caesar [33] is an aspect-oriented language that also supports family polymorphism, permitting application of aspects to mutually recursive nested types.

8. Conclusions

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

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

We have informally described here the static and dynamic semantics of J&. A formal treatment with a proof of soundness can be found in an associated technical report [37].

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

Acknowledgments

Steve Chong, Jed Liu, Ruijie Wang, and Lantian Zheng provided useful feedback on various drafts of this paper. Thank you to Michael Clarkson for his very detailed comments and for the pun. Thanks also to Venugopalan Ramasubramanian for insightful discussions about Pastry and Beehive.

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

References

[1] Alexandre Bergel, Stéphane Ducasse, and Oscar Nierstrasz. Classbox/J: Controlling the scope of change in Java. In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 177–189, San Diego, CA, USA, October 2005.

[2] David Binkley, Susan Horwitz, and Thomas Reps. Program integration for languages with procedure calls. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 4(1):3–35, January 1995.

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

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proc. 5th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

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

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

[7] 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.

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

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

[10] Craig Chambers, David Ungar, Bay-Wei Chang, and Urs Hölzle. Parents are shared parts of objects: Inheritance and encapsulation in Self. *Lisp and Symbolic Computation*, 4(3):207–222, June 1991.

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

[12] Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *Proc. 15th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 130–145, 2000.

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

[14] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.

[15] 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.

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

[17] Erik Ernst. Family polymorphism. In *Proc. 15th European Conference on Object-Oriented Programming (ECOOP)*, LNCS 2072, pages 303–326, 2001.

[18] Erik Ernst. Higher-order hierarchies. In *Proc. 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.

[19] Erik Ernst, Klaus Ostermann, and William R. Cook. A virtual class calculus. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 270–282, Charleston, South Carolina, January 2006.

[20] 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.

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

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

[23] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):345–387, July 1989.

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

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

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

[27] 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.

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

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

[30] Ole Lehrmann Madsen and Birger Møller-Pedersen. Virtual classes: A powerful mechanism for object-oriented programming. In *Proc. 4th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 397–406, October 1989.

[31] Katsuhisa Maruyama and Ken-Ichi Shima. An automatic class generation mechanism by using method integration. *IEEE Transactions on Software Engineering*, 26(5):425–440, May 2000.

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

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

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

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

[36] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. 12th International Compiler Construction Conference (CC'03)*, pages 138–152, April 2003. LNCS 2622.

[37] Nathaniel Nystrom, Xin Qi, and Andrew C. Myers. Nested intersection for scalable software extension, September 2006. `http://www.cs.cornell.edu/nystrom/papers/jet-tr.pdf`.

[38] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proc. 20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, pages 41–57, San Diego, CA, USA, October 2005.

[39] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *Proc. 7th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*,

pages 25–40, October 1992.

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

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

[42] 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 [22], pages 13–23.

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

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

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

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

[47] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

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

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

[50] Alessandro Warth, Milan Stanojević, and Todd Millstein. Statically scoped object adaptation with expanders. In *Proc. 21st ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, Portland, OR, October 2006.

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

programs	$Pr ::= \langle \overline{L}, e \rangle$
class declarations	$L ::= \texttt{class}\ C\ \texttt{extends}\ T\ \{\overline{L}\ \overline{F}\ \overline{M}\}$
field declarations	$F ::= [\texttt{final}]\ T\ f = e$
method declarations	$M ::= T\ m(\overline{T}\ \overline{x})\ \{e\}$
types	$T ::= \circ \mid T.C \mid p.\texttt{class} \mid P\texttt{[}T\texttt{]} \mid \texttt{\&}\overline{T}$
non-dependent types	$S ::= \circ \mid S.C \mid P\texttt{[}S\texttt{]} \mid \texttt{\&}\overline{S}$
classes	$P ::= \circ \mid P.C$
values	$v ::= \texttt{null} \mid \ell$
access paths	$p ::= v \mid x \mid p.f$
expressions	$e ::= v \mid x \mid e.f \mid e_0.f = e_1$
	$\mid e_0.m(\overline{e}) \mid \texttt{new}\ T(\overline{f} = \overline{e}) \mid e_1;\ e_2$
typing environments	$\Gamma ::= \emptyset \mid \Gamma, x\!:\!T \mid \Gamma, \ell\!:\!S \mid \Gamma, p_1 = p_2$

Figure 11. Grammar

A. Formal semantics

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

A grammar for the calculus is shown in Figure 11. Throughout the semantics, we use the notation $\overline{a}$ for the list $a_1, \ldots, a_n$ for $n \geq 0$. The length of $\overline{a}$ is written $\#(\overline{a})$, and the empty list is written nil. We write $\{\overline{a}\}$ for the set containing the members of the list $\overline{a}$. A term with a list subterm should be interpreted as a list of terms; for example, $\overline{f} = \overline{e}$ should be read $f_1 = e_1, \ldots, f_n = e_n$.

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

Following Tribe [11], all classes are nested within a single top-level class $\circ$. Types T are either the top-level class $\circ$, nested classes $T.C$, dependent classes $p.\texttt{class}$, prefix types $P\texttt{[}T\texttt{]}$, or intersection types $\texttt{\&}\overline{T}$. The intersection type $\texttt{\&}\overline{T}$ can be read $T_1 \,\texttt{\&} \cdots \texttt{\&}\, T_n$. A nested class $\circ.C$ of the top-level class may be abbreviated C. Non-dependent types are written S and class names are written P. In the calculus, the prefix type $P\texttt{[}T\texttt{]}$ is well-formed only if some supertype of T is immediately enclosed by a subclass of P. More general prefix types can be constructed by desugaring to this form: for example, if `c` has type `A.B.C`, then `A[c.class]` desugars to `A[A.B[c.class]]`.

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

Type checking is performed in a typing context Γ, which is a list of variable typings $x\!:\!T$, location typings $\ell\!:\!S$, and path equivalence constraints $p_1 = p_2$. Location typings are used to type check the heap during evaluation. Path equivalence constraints are used to assert equivalence of dependent types in the presence of aliasing. They generalize the aliasing facts of the form $\ell_1 = \ell_2.f$ in the Tribe type system [11]. The type-checker does not require an alias analysis be performed; however, the type system could easily be augmented with results of an alias analysis to improve its precision.

A.1 Class lookup

The class table, CT, defined in Figure 12, maps class names P to class declarations. We write $CT(P) = \bot$ if P has no definition. The judgment $\vdash P\ \mathsf{defined}$ states that P is a well-formed class; the judgment holds either when P is a class in the class table or when P further binds a defined class. The mem function returns the set of classes P comprising a non-dependent type S; a type S is equivalent to the intersection of all classes in $\mathsf{mem}(S)$. Using mem, the well-formed class judgment is extended to all non-dependent types S.

A.2 Subclassing and further binding

Inheritance among classes is defined in Figure 12. The rules are similar to those defined for the language Tribe [11]. The judgment $\vdash P \sqsubset_{\mathsf{sc}} P'$ states that P is a declared subclass of P'. The rule SC simply looks up the superclass using the class table CT, substituting the container for `this.class`. Type substitution is defined in Figure 16. By the program well-formedness rules, shown later, the only access path allowed in a superclass declaration is the `this` path, ensuring that the result of substituting for `this` is a non-dependent type.

The judgment $\vdash P_1.C \sqsubset_{\mathsf{fb}} P_2.C$ states that $P_1.C$ further binds $P_2.C$ when P_1 inherits from P_2 and $P_2.C$ is defined. We write $\vdash P_1 \sqsubset P_2$ if P_1 either subclasses or further binds P_2. The reflexive, transitive closure of $\sqsubset$ is $\sqsubset^*$. The relation $\sim$ is an equivalence relation between classes that contain a common nested class C. The function $\mathsf{inh}(S)$ returns the set of all superclasses of S.

A.3 Prefix types

The meaning of non-dependent prefix types $P\texttt{[}S\texttt{]}$ is defined by the prefix function in Figure 12. The P-prefix of a non-dependent type S is the intersection of all classes P' where P and P' transitively share a nested class—that is, P and P' are equivalent under the $\sim$ relation—and S extends nested classes of both P and P'. The intuition behind the definition is that S extends some class that is contained in the intersection of P and P'. This definition ensures that if P is a subtype of P', then $P\texttt{[}S\texttt{]}$ is equal to $P'\texttt{[}S\texttt{]}$, as desired in Section 3.5.

$$\boxed{CT(P)}$$

$$\frac{Pr = \langle \overline{L}, e \rangle}{CT(\circ) = \mathtt{class}\ \circ\ \mathtt{extends}\ \&\mathsf{nil}\ \{\overline{L}\}}$$

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

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

$$\vdash \circ\ \mathsf{defined} \qquad (\textsc{Def-outer})$$

$$\frac{CT(P) \neq \bot}{\vdash P\ \mathsf{defined}} \qquad (\textsc{Def-expl})$$

$$\frac{\vdash P\ \mathsf{defined} \quad \vdash P \sqsubset P' \quad \vdash P'.C\ \mathsf{defined}}{\vdash P.C\ \mathsf{defined}} \qquad (\textsc{Def-inh})$$

$$\frac{\vdash S\ \mathsf{defined} \quad P \in \mathsf{mem}(S) \quad \vdash P.C\ \mathsf{defined}}{\vdash S.C\ \mathsf{defined}} \qquad (\textsc{Def-nest})$$

$$\frac{\vdash S_i\ \mathsf{defined}^{\forall S_i \in \overline{S}}}{\vdash \&\overline{S}\ \mathsf{defined}} \qquad (\textsc{Def-meet})$$

$$\boxed{\vdash P_1 \sqsubset_{\mathsf{sc}} P_2}$$

$$\frac{\begin{array}{c} \vdash P_1 \sqsubset^* P \\ CT(P.C) = \mathtt{class}\ C\ \mathtt{extends}\ T\ \{\ldots\} \\ T\{\!\{\emptyset;\ P_1/\mathtt{this}\}\!\} = S \\ P_2 \in \mathsf{mem}(S) \end{array}}{\vdash P_1.C \sqsubset_{\mathsf{sc}} P_2} \qquad (\text{SC})$$

$$\boxed{\vdash P_1 \sqsubset_{\mathsf{fb}} P_2}$$

$$\frac{\vdash P_1 \sqsubset P_2 \quad \vdash P_2.C\ \mathsf{defined}}{\vdash P_1.C \sqsubset_{\mathsf{fb}} P_2.C} \qquad (\text{FB})$$

$$\boxed{\vdash P_1 \sqsubset P_2}$$

$$\frac{\vdash P_1 \sqsubset_{\mathsf{sc}} P_2}{\vdash P_1 \sqsubset P_2}\ (\textsc{Inh-SC}) \qquad \frac{\vdash P_1 \sqsubset_{\mathsf{fb}} P_2}{\vdash P_1 \sqsubset P_2}\ (\textsc{Inh-FB})$$

$$\boxed{\vdash P_1 \sim P_2}$$

$$\frac{\vdash P_1.C \sqsubset_{\mathsf{fb}} P_0.C \quad \vdash P_2.C \sqsubset_{\mathsf{fb}} P_0.C}{\vdash P_1 \sim P_2} \qquad (\textsc{Rel-fb})$$

$$\vdash P \sim P\ (\textsc{Rel-refl}) \qquad \frac{\vdash P_1 \sim P_2}{\vdash P_2 \sim P_1}\ (\textsc{Rel-sym}) \qquad \frac{\vdash P_1 \sim P_2 \quad \vdash P_2 \sim P_3}{\vdash P_1 \sim P_3}\ (\textsc{Rel-trans})$$

$$\boxed{\mathsf{mem}(S)}$$

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

$$\frac{D = \{P_i \in \mathsf{mem}(S) \mid \vdash P_i.C\ \mathsf{defined}\}}{\mathsf{mem}(S.C) = \bigcup_{P_i \in D} P_i.C}$$

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

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

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

$$\begin{array}{l} \mathsf{prefix}(P,S) = \&\{P' \mid \exists C, C'. \\ \qquad\qquad \vdash P \sim P' \\ \qquad\qquad \wedge P.C \in \mathsf{inh}(S) \\ \qquad\qquad \wedge P'.C' \in \mathsf{inh}(S)\} \end{array}$$

Figure 12. Subclassing and auxiliary functions

A.4 Member lookup

Method and field lookup functions are shown in Figure 14. For a class P, we define $\mathsf{ownFields}(P)$ and $\mathsf{ownMethods}(P)$ to be the set of fields and methods declared in the class. Using these definitions, the set of fields and methods declared or inherited by a non-dependent type S is defined by the $\mathsf{fields}(S)$ and $\mathsf{methods}(S)$ functions. The function fnames returns the set of field names for a list of fields $\overline{F}$. The ftype function returns the declared type of a field f of an arbitrary type T in environment Γ. The mtype function provides similar functionality for methods.

The method body for a method m in type S is returned by mbody. For simplicity, the formal semantics presented here do not specify what method body to dispatch to when one method overrides another; precise specification of method dispatch is not necessary to prove soundness of the type system.

A.5 Exactness

Before proceeding, some auxiliary functions need to be defined. The function $\mathsf{paths}(T)$ returns the set of access paths in the structure of type T. The function $\mathsf{exacts}(T)$ returns the set of (maximal) exact types embedded in the structure of type T. The function $\mathsf{prefixExact}_k(T)$ is true if the kth prefix of T is an exact type. If $\mathsf{prefixExact}_k(T)$, then necessarily $\mathsf{prefixExact}_{k+1}(T)$.

A.6 Simple bounds

The judgment $\Gamma \vdash T \trianglelefteq S$ in Figure 15 states that T has a non-dependent bounding type S. For dependent classes $p.\mathtt{class}$, the bounding type is simply the bound on the declared type of p. For prefix types $P\mathtt{[}T\mathtt{]}$, the bound is the result of computing the prefix function for P and the bounding type of T.

A.7 Type well-formedness

A type T is well-formed in a context Γ is written $\Gamma \vdash T$. A class P is well-formed if it is in the class table CT. A nested type $T.C$ is well-formed if T is well-formed and has bound S and if $S.C$ is defined. A dependent class $p.\mathtt{class}$ is well-formed if p is a final access path. A prefix type $P\mathtt{[}T\mathtt{]}$ is well-formed if T has simple bound S and $\mathsf{prefix}(P,S)$ is not empty; in other words, there is some superclass of T whose enclosing class is related to P by further binding. Finally an intersection type $\&\overline{T}$ is well-formed if the following three conditions hold:

- All constituent types T_i are well-formed, and
- All exact types in the structure of $\&\overline{T}$ are equivalent up to aliasing, ensuring they all refer to the same run-time class.
- All constituent types T_i have the same level of exactness. This condition ensures that the intersection has the same level of exactness as any one of its constituents.

$$\begin{aligned}
\mathsf{paths}(\circ) &= \emptyset \\
\mathsf{paths}(T.C) &= \mathsf{paths}(T) \\
\mathsf{paths}(p.\texttt{class}) &= \{p\} \\
\mathsf{paths}(P\texttt{[}T\texttt{]}) &= \mathsf{paths}(T) \\
\mathsf{paths}(\texttt{\&}\overline{T}) &= \bigcup_{T_i \in \overline{T}} \mathsf{paths}(T_i)
\end{aligned}$$

$$\mathsf{exacts}(T) = \begin{cases}
\{T\} & \text{if } \mathsf{exact}(T) \\
\emptyset & \text{if } T = \circ \\
\mathsf{exacts}(T_0) & \text{if } \neg\mathsf{exact}(T) \text{ and } T = T_0.C \\
\mathsf{exacts}(T_0) & \text{if } \neg\mathsf{exact}(T) \text{ and } T = P\texttt{[}T_0\texttt{]} \\
\bigcup_{T_i \in \overline{T}} \mathsf{exacts}(T_i) & \text{if } \neg\mathsf{exact}(T) \text{ and } T = \texttt{\&}\overline{T}
\end{cases}$$

$$\begin{aligned}
\mathsf{exact}(T) &= \mathsf{prefixExact}_0(T) \\
\mathsf{prefixExact}_k(\circ) &= \mathsf{false} \\
\mathsf{prefixExact}_k(T.C) &= \begin{cases} \mathsf{false} & \text{if } k = 0 \\ \mathsf{prefixExact}_{k-1}(T) & \text{otherwise} \end{cases} \\
\mathsf{prefixExact}_k(p.\texttt{class}) &= \mathsf{true} \\
\mathsf{prefixExact}_k(P\texttt{[}T\texttt{]}) &= \mathsf{prefixExact}_{k+1}(T) \\
\mathsf{prefixExact}_k(\texttt{\&}\overline{T}) &= \bigvee_{T_i \in \overline{T}} \mathsf{prefixExact}_k(T_i)
\end{aligned}$$

Figure 13. Auxiliary functions

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

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

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

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

$$\frac{\overline{F} = [\texttt{final}]\ \overline{T}\ \overline{f} = \overline{e}}{\mathsf{fnames}(\overline{F}) = \{\overline{f}\}}$$

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

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

$$\frac{\begin{array}{c}\Gamma \vdash T \trianglelefteq S \\ \mathsf{methods}(S) = \overline{M} \\ M_i = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}\end{array}}{\mathsf{mtype}(\Gamma, T, m) = (\overline{x}\!:\!\overline{T}) \to T_{n+1}}$$

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

Figure 14. Member lookup

A.8 Type substitution

The rules for type substitution are shown in Figure 16. The function $T\{\!\{\Gamma;\ T_x/x\}\!\}$ substitutes T_x for x in T. The environment Γ is used to look up field types when substituting a non-dependent class into a field-path dependent class. T_x should be well-formed in Γ and a subtype of x's declared type.

A.9 Final access paths

The judgment $\Gamma \vdash_{\mathsf{final}} p : T$ in Figure 15 states that the access path p is a well-typed final access path in context Γ. The `null` path can take on any non-dependent type. A location path ℓ has the type declared in the environment. A variable path x has the type declared in the environment. Finally a field path $p.f$ is final if p is final with type T, and the type of the field path is determined by looking up the field type.

A.10 Typing

For arbitrary expressions, the judgment $\Gamma \vdash e : T$ states that e has type T in context Γ.

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

$$\Gamma \vdash P \trianglelefteq P\ \textsc{(Bd-simp)} \qquad \frac{\Gamma \vdash T \trianglelefteq S}{\Gamma \vdash T.C \trianglelefteq S.C}\ \textsc{(Bd-nest)} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{final}} p : T \\ \Gamma \vdash T \trianglelefteq S\end{array}}{\Gamma \vdash p.\mathtt{class} \trianglelefteq S}\ \textsc{(Bd-fin)} \qquad \frac{\begin{array}{c}\Gamma \vdash T \trianglelefteq S \\ \mathsf{prefix}(P, S) = S'\end{array}}{\Gamma \vdash P\mathtt{[}T\mathtt{]} \trianglelefteq S'}\ \textsc{(Bd-pre)} \qquad \frac{\Gamma \vdash T_i \trianglelefteq S_i{}^{\forall i}}{\Gamma \vdash \&\overline{T} \trianglelefteq \&\overline{S}}\ \textsc{(Bd-meet)}$$

$\boxed{\Gamma \vdash T}$

$$\frac{CT(P) \neq \bot}{\Gamma \vdash P}\ \textsc{(WF-simp)} \qquad \frac{\begin{array}{c}\Gamma \vdash T \\ \Gamma \vdash T \trianglelefteq S \\ \vdash S.C\ \mathsf{defined}\end{array}}{\Gamma \vdash T.C}\ \textsc{(WF-nest)} \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p.\mathtt{class}}\ \textsc{(WF-fin)} \qquad \frac{\begin{array}{c}\Gamma \vdash P \quad \Gamma \vdash T \\ \Gamma \vdash P\mathtt{[}T\mathtt{]} \trianglelefteq S \\ S \neq \&\mathsf{nil}\end{array}}{\Gamma \vdash P\mathtt{[}T\mathtt{]}}\ \textsc{(WF-pre)} \qquad \frac{\begin{array}{c}\Gamma \vdash T_i{}^{\forall i} \\ \Gamma \vdash T_i \simeq T_j{}^{\forall T_i, T_j \in \mathsf{exacts}(\&\overline{T})} \\ \mathsf{prefixExact}_k(T_i) \Rightarrow \mathsf{prefixExact}_k(T_j)^{\forall T_i, T_j \in \overline{T}}\end{array}}{\Gamma \vdash \&\overline{T}}\ \textsc{(WF-meet)}$$

$\boxed{\Gamma \vdash T_1 \simeq T_2}$

$$\Gamma \vdash T \simeq T \qquad \frac{\Gamma \vdash T_2 \simeq T_1}{\Gamma \vdash T_1 \simeq T_2} \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 \simeq T_2 \\ \Gamma \vdash T_2 \simeq T_3\end{array}}{\Gamma \vdash T_1 \simeq T_3} \qquad \frac{\Gamma \vdash p_1 = p_2}{\Gamma \vdash TE[p_1] \simeq TE[p_2]}$$

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

$$\frac{\Gamma \vdash S}{\Gamma \vdash_{\mathsf{final}} \mathtt{null} : S}\ \textsc{(F-null)} \qquad \frac{\ell : S \in \Gamma}{\Gamma \vdash_{\mathsf{final}} \ell : S}\ \textsc{(F-loc)} \qquad \frac{x : T \in \Gamma}{\Gamma \vdash_{\mathsf{final}} x : T}\ \textsc{(F-var)} \qquad \frac{\begin{array}{c}\Gamma \vdash_{\mathsf{final}} p : T \\ \mathsf{ftype}(\Gamma, T, f) = \mathtt{final}\ T_f\end{array}}{\Gamma \vdash_{\mathsf{final}} p.f : T_f}\ \textsc{(F-get)}$$

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

$$\frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p : p.\mathtt{class}}\ \textsc{(T-fin)} \qquad \frac{\begin{array}{c}\Gamma \vdash e : T \\ \mathsf{ftype}(\Gamma, T, f) = [\mathtt{final}]\ T_f\end{array}}{\Gamma \vdash e.f : T_f}\ \textsc{(T-get)} \qquad \frac{\begin{array}{c}\Gamma \vdash e_0 : T_0 \quad \Gamma \vdash e_1 : T_f \\ \mathsf{ftype}(\Gamma, T_0, f) = T_f\end{array}}{\Gamma \vdash e_0.f = e_1 : T_f}\ \textsc{(T-set)} \qquad \frac{\Gamma \vdash e_1 \vdash T_1 \quad \Gamma \vdash e_2 \vdash T_2}{\Gamma \vdash e_1;\ e_2 : T_2}\ \textsc{(T-seq)}$$

$$\frac{\begin{array}{c}\Gamma \vdash T \quad \Gamma \vdash \overline{e} : \overline{T} \\ \mathsf{ftype}(\Gamma, T, f_i) = [\mathtt{final}]\ T_i{}^{\forall f_i \in \overline{f}}\end{array}}{\Gamma \vdash \mathtt{new}\ T(\overline{f} = \overline{e}) : T}\ \textsc{(T-new)} \qquad \frac{\Gamma \vdash e : T \quad \Gamma \vdash T \leq T'}{\Gamma \vdash e : T'}\ \textsc{(T-sub)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : T_0^0 \quad \Gamma \vdash e_i : T_i^{i\ \forall i = 1..n} \\ n = \#(\overline{e}) = \#(\overline{x}) \quad x_0 = \mathtt{this} \\ \mathsf{mtype}(\Gamma, T_0^0, m) = (\overline{x} : \overline{T^0}) \to T_{n+1}^0 \\ T_i^{j-1}\{\!\!\{\Gamma;\ T_{j-1}^{j-1}/x_{j-1}\}\!\!\} = T_i^{j\ \forall i \in 1..n+1, j \in 1..i} \\ \mathsf{prefixExact}_k(T_i^{j-1}) \Rightarrow \mathsf{prefixExact}_k(T_i^j)^{\forall i \in 1..n, j \in 1..i} \\ p.f \in \mathsf{paths}(T_i^{j-1}) \Rightarrow p' \in \mathsf{paths}(T_i^j) \wedge \Gamma \vdash p' = p\{e_{j-1}/x_{j-1}\}.f^{\forall i \in 1..n+1, j \in 1..i}\end{array}}{\Gamma \vdash e_0.m(\overline{e}) : T_{n+1}^{n+1}}\ \textsc{(T-call)}$$

$\boxed{\Gamma \vdash p_1 = p_2}$

$$\frac{p_1 = p_2 \in \Gamma}{\Gamma \vdash p_1 = p_2}\ \textsc{(A-env)} \qquad \frac{\begin{array}{c}\Gamma \vdash p_1 = p_2 \\ \Gamma \vdash_{\mathsf{final}} p_1 : T_1 \\ \Gamma \vdash_{\mathsf{final}} p_1 : T_2\end{array}}{\Gamma \vdash p_1.f = p_2.f}\ \textsc{(A-field)} \qquad \Gamma \vdash p = p\ \textsc{(A-refl)} \qquad \frac{\Gamma \vdash p_2 = p_1}{\Gamma \vdash p_1 = p_2}\ \textsc{(A-sym)} \qquad \frac{\begin{array}{c}\Gamma \vdash p_1 = p_2 \\ \Gamma \vdash p_2 = p_3\end{array}}{\Gamma \vdash p_1 = p_3}\ \textsc{(A-trans)}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash T_1 \leq T_2 \\ \Gamma \vdash T_2 \leq T_3\end{array}}{\Gamma \vdash T_1 \leq T_3}\ \textsc{(S-trans)} \qquad \frac{\begin{array}{c}\Gamma \vdash T \leq P \\ CT(P.C) = \mathtt{class}\ C\ \mathtt{extends}\ T'\ \{\ldots\} \\ T'\{\!\!\{\Gamma;\ T/\mathtt{this}\}\!\!\} = T''\end{array}}{\Gamma \vdash T.C \leq T''}\ \textsc{(S-sup)} \qquad \frac{\Gamma \vdash T \quad \Gamma \vdash T \trianglelefteq S}{\Gamma \vdash T \leq S}\ \textsc{(S-bound)}$$

$$\frac{\Gamma \vdash T_1 \leq T_2 \quad \Gamma \vdash T_2.C}{\Gamma \vdash T_1.C \leq T_2.C}\ \textsc{(S-nest)} \qquad \frac{\Gamma \vdash_{\mathsf{final}} p : T}{\Gamma \vdash p.\mathtt{class} \leq T}\ \textsc{(S-fin)} \qquad \frac{\begin{array}{c}\Gamma \vdash T_1 \leq T_2 \\ \Gamma \vdash P\mathtt{[}T_2\mathtt{]}\end{array}}{\Gamma \vdash P\mathtt{[}T_1\mathtt{]} \leq P\mathtt{[}T_2\mathtt{]}}\ \textsc{(S-pre-s1)} \qquad \frac{\begin{array}{c}\vdash P_1 \sim P_2 \vee \vdash P_1 \sqsubset P_2 \\ \Gamma \vdash P_1\mathtt{[}T\mathtt{]} \\ \Gamma \vdash P_2\mathtt{[}T\mathtt{]}\end{array}}{\Gamma \vdash P_1\mathtt{[}T\mathtt{]} \approx P_2\mathtt{[}T\mathtt{]}}\ \textsc{(S-pre-s2)}$$

$$\frac{\Gamma \vdash T \leq P.C}{\Gamma \vdash T \leq P\mathtt{[}T\mathtt{]}.C}\ \textsc{(S-out)} \qquad \frac{\Gamma \vdash P\mathtt{[}T.C\mathtt{]}}{\Gamma \vdash T \approx P\mathtt{[}T.C\mathtt{]}}\ \textsc{(S-in)} \qquad \Gamma \vdash \&\overline{T} \leq T_i\ \textsc{(S-meet-lb)} \qquad \frac{\Gamma \vdash T \leq T_i{}^{\forall i}}{\Gamma \vdash T \leq \&\overline{T}}\ \textsc{(S-meet-g)}$$

$$\frac{\Gamma \vdash T_1 \simeq T_2}{\Gamma \vdash T_1 \approx T_2}\ \textsc{(S-alias)} \qquad \frac{\begin{array}{c}\Gamma \vdash U_1 \trianglelefteq S_1 \quad \Gamma \vdash U_1 \quad \mathsf{exact}(U_1) \\ \Gamma \vdash U_2 \trianglelefteq S_2 \quad \Gamma \vdash U_2 \quad \emptyset \vdash S_1 \approx S_2\end{array}}{\Gamma \vdash U_1 \leq U_2}\ \textsc{(S-eval)} \qquad \frac{\Gamma \vdash T_1 \leq T_2 \quad \mathsf{exact}(T_2)}{\Gamma \vdash T_1 \approx T_2}\ \textsc{(S-exact)}$$

Figure 15. Static semantics

$\boxed{T\{\!\{\Gamma;\ T_x/x\}\!\}}$

$$\circ\{\!\{\Gamma;\ T_x/x\}\!\} = \circ$$

$$T.C\{\!\{\Gamma;\ T_x/x\}\!\} = T\{\!\{\Gamma;\ T_x/x\}\!\}.C$$

$$v.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = v.\texttt{class}$$

$$\frac{x \neq y}{y.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = y.\texttt{class}}$$

$$x.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_x$$

$$\frac{p.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = p'.\texttt{class}}{p.f.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = p'.f.\texttt{class}}$$

$$\frac{\begin{array}{c} p.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_p \\ T_p \neq p'.\texttt{class} \\ \mathsf{ftype}(\Gamma, T_p, f) = T_f \end{array}}{p.f.\texttt{class}\{\!\{\Gamma;\ T_x/x\}\!\} = T_f}$$

$$\frac{T\{\!\{\Gamma;\ T_x/x\}\!\} = T'}{P\texttt{[}T\texttt{]}\{\!\{\Gamma;\ T_x/x\}\!\} = P\texttt{[}T'\texttt{]}}$$

$$\frac{\overline{T_i\{\!\{\Gamma;\ T_x/x\}\!\} = T_i'}^{\,\forall i}}{\texttt{\&}\overline{T}\{\!\{\Gamma;\ T_x/x\}\!\} = \texttt{\&}\overline{T'}}$$

Figure 16. Type substitution

Any final access path p has type $p.\texttt{class}$ by T-FIN. The subtyping rule S-FIN and subsumption (T-SUB) give the standard typing rules for values and parameters x.

The type of a field access $e.f$ is obtained by looking up the field in T, the static type of e. The rule T-SET checks if the expression being assigned from has the same type.

By T-SEQ, a sequence expression takes the type of the second expression in the sequence.

A `new` expression is well-typed via T-NEW if it initializes only declared fields of a well-formed type.

Calls are checked with T-CALL by looking up the method type, then substituting in the receiver type and the actual argument types for `this` and the formal parameters. Type substitution is defined in Figure 16. A type T is exact, written $\mathsf{exact}(T)$, if it is a dependent class, a prefix of an exact type, or an intersection containing an exact type. The function $\mathsf{exact}(T)$ is defined in Figure using the function $\mathsf{prefixExact}_k(T)$, which returns whether the kth enclosing prefix of T is exact. Substituting into the The actuals must have the same type as the substituted formal types. To ensure subtyping is preserved by the substitution, substitution must preserve prefix exactness. This ensures that if the type of formal i is dependent on `this` or on another formal j, the ith actual value has a type dependent on the actual receiver or on actual j. Substitution of the return type need not preserve exactness.

Substitution must also preserve field paths. Two different objects used as actuals may have the same dependent type, but may contain final fields that point to objects of different clases. Preserving field paths ensures that the substituted field path is dependent on the actual target, not on another object of the same class which may have initialized the field differently.

Finally, T-SUB is the standard subsumption rule.

A.11 Subtyping and type equivalence

Subtyping rules are defined in Figure 15. The judgment $\Gamma \vdash T \leq T'$ states that T is a subtype of T' in context Γ. The rules ensure that syntactically different types representing the same sets of values are considered equal. The judgment $\Gamma \vdash T \approx T'$ is sugar for the pair of judgments $\Gamma \vdash T \leq T'$ and $\Gamma \vdash T' \leq T$.

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

S-BOUND states that a type is a subtype of its bounding simple type. The rule S-NEST states that a nested class C is covariant with its containing class; that is, further binding implies subtyping. S-FIN states that a dependent class is a subtype of its declared bound; with F-NULL, this rule also implies that `null.class` is a subtype of any well-formed simple type.

Subtyping of prefix types is covariant by the rules S-PRE-S1 and S-PRE-S2. S-OUT and S-IN, and relate prefix types to non-prefix types.

S-MEET-LB and S-MEET-G are from Compagnoni and Pierce [13] and define subtyping for intersection types. Together these two rules imply that intersection types are associative and commutative and that the singleton intersection type $\texttt{\&}T$ is equivalent to its element type T. With the other rules above, these rules also imply the intuitive judgments $\Gamma \vdash P\texttt{[}\texttt{\&}\overline{T}\texttt{]} \leq P\texttt{[}T_i\texttt{]}$ and $\Gamma \vdash (\texttt{\&}\overline{T}).C \leq T_i.C$.

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

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

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

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

$$\frac{\vdash \Gamma\ \mathsf{ok} \quad \Gamma \vdash_{\mathsf{final}} p_1 \quad \Gamma \vdash_{\mathsf{final}} p_2}{\vdash \Gamma, p_1 = p_2\ \mathsf{ok}}$$

Figure 17. Well-formed environments

$\boxed{\Gamma\{v/x\}}$

$$\begin{aligned}
\emptyset\{v/x\} &= \emptyset \\
(\Gamma, x\!:\!T)\{v/x\} &= \Gamma \\
(\Gamma, y\!:\!T)\{v/x\} &= \Gamma\{v/x\}, y\!:\!T\{v/x\} \\
(\Gamma, \ell\!:\!S)\{v/x\} &= \Gamma\{v/x\}, \ell\!:\!S \\
(\Gamma, p_1 = p_2)\{v/x\} &= \Gamma\{v/x\}, p_1\{v/x\} = p_2\{v/x\}
\end{aligned}$$

Figure 18. Environment substitution

Finally, the rule S-EVAL states that a fully evaluated type (i.e., a type containing only value paths) is a supertype of any fully evaluated exact type with the same bounding type. This rule ensures, for example, that ℓ_1.`class` $\approx$ ℓ_2.`class` if ℓ_1 and ℓ_2 both point to objects of the same type.

A.12 Program typing

Program typing rules are presented in Figure 19. The P-OK says the program *Pr* is well-formed if all class declarations are well-formed, if the "`main`" expression is well-typed, and if the transitive closure of the inheritance relation $\sqsubset$ is acyclic.

By L-OK, a class declaration is well-formed if all its members are well-formed and its superclass is well-formed in an environment containing only `this` bound to the class's container. Additionally, the only access path embedded in the superclass declaration can be `this`. The class must also conform to all of its superclasses.

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

- If both P and P' have a member class D, then $P.D$'s declared superclass is a subtype of $P'.D$'s.
- The field names of P and P' are disjoint. This requirement simplifies the semantics by ensuring field names are unique.
- If both P and P' define a method m, then the method in P correctly overrides the method P'.

Method M in P correctly overrides M' if the number of formal parameters are equal, the parameter types of M are supertypes of the parameter types of M', and the return type of M is a subtype of M'. Subtyping checks are done with fresh names substituted in for the parameter names occurring in the types. Using the judgment $\vdash \Gamma\ \mathsf{ok}$, it is required that the type of formal parameter i depends only on `this` and formal parameters 1 through $i-1$.

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

A.13 Environments

Environments are well-formed by the judgment $\vdash \Gamma\ \mathsf{ok}$, defined in Figure 17. Environment substitution is defined in Figure 18.

A.14 Operational semantics

A small-step operational semantics is shown in Figure 20. The semantics are defined using a reduction relation $\longrightarrow$, which maps a configuration of an expression e and a heap H to a new configuration. A heap H is a function from memory locations ℓ to objects $S\ \{\overline{f} = \overline{v}\}$. The notation $e, H \longrightarrow r, H'$ means that expression e and heap H step to result r and heap H'. Results are either expressions or $\mathsf{NullError}$. The initial configuration for program $\langle \overline{L}, e\rangle$ is $e, \emptyset$. Final configurations are of the form v, H or $\mathsf{NullError}, H$.

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

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

$$\frac{\begin{array}{c} T \neq \circ \\ \mathtt{this}\!:\!P \vdash T \\ \mathsf{paths}(T) \subseteq \{\mathtt{this}\} \\ \neg\mathsf{exact}(T) \end{array}}{P \vdash T\ \mathsf{super\ ok}}$$

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

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

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

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

Figure 19. Program typing

We write $H[\ell := o]$ for H with $H(\ell)$ remapped to o, that is:

$$\begin{aligned} \emptyset[\ell := o] &= \ell \mapsto o \\ (H, \ell \mapsto o')[\ell := o] &= H, \ell \mapsto o \\ (H, \ell' \mapsto o')[\ell := o] &= H[\ell := o], \ell' \mapsto o' \end{aligned}$$

To account for aliasing of access paths, typing environments include path equivalence constraints $p_1 = p_2$. The function $H^{\dagger}$ returns a typing context constructed from a heap H by inserting location types and aliasing information for fields into the environment.

$$\begin{aligned} \emptyset^{\dagger} &= \emptyset \\ (H, \ell \mapsto S\ \{\overline{f} = \overline{v}\})^{\dagger} &= H^{\dagger}, \ell\!:\!S, \ell.\overline{f'} = \overline{v'} \\ \text{where } \overline{f'} &= \{f_i \in \overline{f} \mid \mathsf{ftype}(\emptyset, S, f_i) = \mathtt{final}\ T_i\} \end{aligned}$$

The equivalence constraints ensure that when $\ell_1.f, H \longrightarrow \ell_2$, $\ell_2.\mathtt{class}$ is a subtype of $\ell_1.f.\mathtt{class}$. This is essential for proving type preservation.

The reduction rules are mostly straightforward. Order of evaluation is captured by an evaluation context E (an expression with a hole $[\cdot]$) and the congruence rule R-CONG. Since types are dependent, expressions used in types must be evaluated as well. We write U for a type containing no redex. The rule R-NULL propagates a dereference of a `null` pointer out through the evaluation contexts to produce a NullError, simulating a Java `NullPointerException`.

R-GET and R-SET get and set a field in a heap object, respectively. R-CALL uses the mbody function defined in Figure 14 to locate the most specific implementation of method m.

$$
\begin{array}{lrl}
\text{objects} & o & ::= S\,\{\overline{f}=\overline{v}\} \\
\text{heaps} & H & ::= \emptyset \mid H,\ell \mapsto o \\
\text{results} & r & ::= e \mid \mathsf{NullError} \\
\text{evaluated types} & U & ::= \circ \mid U.C \mid \ell.\mathtt{class} \mid P\mathtt{[}U\mathtt{]} \mid \mathtt{\&}\overline{U} \\
\text{evaluation contexts} & E & ::= [\cdot] \\
 & & \quad\mid E.f \\
 & & \quad\mid \mathtt{new}\ TE(\overline{f}=\overline{e}) \\
 & & \quad\mid \mathtt{new}\ U(\overline{f}=\overline{v}, f=E, \overline{f'}=\overline{e}) \\
 & & \quad\mid E.f=e \\
 & & \quad\mid \ell.f=E \\
 & & \quad\mid E.m(\overline{e}) \\
 & & \quad\mid \ell.m(\overline{v},E,\overline{e}) \\
 & & \quad\mid E;\ e \\
\text{type evaluation contexts} & TE & ::= TE.C \\
 & & \quad\mid E.\mathtt{class} \\
 & & \quad\mid P\mathtt{[}TE\mathtt{]} \\
 & & \quad\mid \mathtt{\&}(\overline{U},TE,\overline{T}) \\
\text{null error contexts} & NE & ::= \mathtt{null}.f \\
 & & \quad\mid \mathtt{null}.f=e \\
 & & \quad\mid \mathtt{null}.m(\overline{e}) \\
 & & \quad\mid \mathtt{new}\ TE[\mathtt{null}](\overline{f}=\overline{e}) \\
 & & \quad\mid \mathsf{NullError}
\end{array}
$$

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

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

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

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

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

$$\frac{\ell\!:\!S \in H^{\dagger} \quad \mathsf{mbody}(S,m)=T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\} \quad n=\#(\overline{v})=\#(\overline{x})}{\ell.m(\overline{v}),H \longrightarrow e\{\ell/\mathtt{this},\overline{v}/\overline{x}\},H} \qquad \text{(R-CALL)}$$

$$\frac{\begin{array}{c}H^{\dagger}\vdash U \trianglelefteq S\\ \mathsf{fnames}(\mathsf{fields}(S))=\overline{f}\cup\overline{f'} \quad \#(\overline{f'})\neq 0\\ \mathsf{finit}(S,\overline{f'})=\overline{e'}\end{array}}{\mathtt{new}\ U(\overline{f}=\overline{v}),H \longrightarrow \mathtt{new}\ U(\overline{f}=\overline{v},\overline{f'}=\overline{e'}),H} \qquad \text{(R-NEW)}$$

$$\frac{\begin{array}{c}H^{\dagger}\vdash U \trianglelefteq S \quad \{\overline{f}\}=\mathsf{fnames}(\mathsf{fields}(S))\\ \ell\notin \mathrm{dom}(H) \quad H'=H[\ell:=S\,\{\overline{f}=\overline{v}\}]\end{array}}{\mathtt{new}\ U(\overline{f}=\overline{v}),H \longrightarrow \ell,H'} \qquad \text{(R-ALLOC)}$$

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

Figure 20. Operational semantics

$$\frac{\begin{array}{c}H(\ell)=S\,\{\overline{f}=\overline{v}\}\\ \mathsf{fields}(S)=[\mathtt{final}]\ \overline{T}\ \overline{f}=\overline{e}\\ H^{\dagger}\vdash\overline{v}\!:\!\overline{T}\\ \overline{v}\subseteq\mathrm{dom}(H)\cup\{\mathtt{null}\}\end{array}}{H\vdash\ell} \qquad \text{(H-LOC)}$$

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

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

Figure 21. Well-formed heaps

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

A.15 Well-formed heaps

Figure 21 shows the heap typing rules. The judgment $H \vdash \ell$ states that a location ℓ is well-formed for a heap H if it maps to an object of type S containing all declared fields of S and each value stored in those fields has the correct type and, if a location, is also well-formed in H. Rule H-NULL states that the `null` value is always well-formed.

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

B. Soundness

To prove soundness we use the standard technique of proving subject reduction and progress lemmas [51].

B.1 Substitution

This is a useful pair of lemmas that allows many of the type substitution (T for x) lemmas proved above (XXX below) to be used easily to prove value substitution (v for x) lemmas.

LEMMA B.1. *If* $x\!:\!T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v\!:\!T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash p.\texttt{class}$, *then* $p.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = p\{v/x\}.\texttt{class}$.

PROOF: By structual induction on p. Let $p' = p\{v/x\}$.

- If $p = x$, then $p' = v$. The case follows trivially since $x.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = v.\texttt{class}$.
- If $p = p_0.f$, then $p' = p_0\{v/x\}.f$. By the induction hypothesis, $p_0.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = p_0\{v/x\}.\texttt{class}$. Thus, by the definition of type substitution, $p_0.f.\texttt{class}\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = p_0\{v/x\}.f.\texttt{class}$.
- Otherwise, $p' = p$ and the case holds trivially.

□

LEMMA B.2. *If* $x\!:\!T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v\!:\!T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash T$, *then* $T\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = T\{v/x\}$.

PROOF: By structural induction on T.

- $T = \circ$. Trivial.
- $T = T_0.C$. Follows from the induction hypothesis and definition of type substitution.
- $T = p.\texttt{class}$. Then $T\{v/x\} = p'.\texttt{class}$ where $p' = p\{v/x\}$. The case follows from Lemma B.1.
- $T = P\,\texttt{[}T_0\texttt{]}$. Then $T\{v/x\} = P\,\texttt{[}T_0\{v/x\}\texttt{]}$.

  By the induction hypothesis, $T_0\{\!\{\Gamma;\ v.\texttt{class}/x\}\!\} = T_0\{v/x\}$. Since $\mathsf{exact}(P\,\texttt{[}T_0\texttt{]})$, we also have $\mathsf{exact}(P\,\texttt{[}T_0\{v/x\}\texttt{]})$.

  Hence, the case holds by the definition of type substitution,
- $T = \texttt{\&}\overline{T}$. Follows from the induction hypothesis and definition of type substitution.

□

LEMMA B.3. *If* $x\!:\!T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v\!:\!T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash_{\mathsf{final}} p\!:\!T$, *then* $\Gamma\{v/x\} \vdash_{\mathsf{final}} p\{v/x\}\!:\!T_s$. *where* $\Gamma\{v/x\} \vdash T_s \leq T\{v/x\}$.

PROOF: By induction on the derivation of $\Gamma \vdash_{\mathsf{final}} p\!:\!T$. Let $p' = p\{v/x\}$

- F-NULL. Then $p = p'$. By F-NULL, $\Gamma\{v/x\} \vdash_{\mathsf{final}} \texttt{null}\!:\!T\{v/x\}$.
- F-LOC. Then $p = p'$ and $T = T\{v/x\}$.
- F-VAR. Let $p = y \neq x$. Then $p = p'$. Then $y\!:\!T \in \Gamma$. If x is not free in T, then $T\{v/x\} = T$. If, on the other hand, x is free in T, then $y\!:\!T\{v/x\} \in \Gamma\{v/x\}$ and we can derive $\Gamma\{v/x\} \vdash_{\mathsf{final}} y\!:\!T\{v/x\}$ by F-VAR.

  Now, let $p = x$. Then $p' = v$ and $T = T_x$ and $T\{v/x\} = T_s = T_v$. Since we assumed $\Gamma\{v/x\} \vdash T_v \leq T_x$, the case holds trivially.
- F-GET. Then $p = p_0.f$, $\Gamma \vdash_{\mathsf{final}} p_0\!:\!T_0$, $\mathsf{ftype}(\Gamma, T_0, f) = T_f$, and $T = T_f$.

  By the induction hypothesis, $\Gamma\{v/x\} \vdash_{\mathsf{final}} p_0\{v/x\}\!:\!T_0'$, where $\Gamma\{v/x\} \vdash T_0' \leq T_0\{v/x\}$.

  By Lemma B.6, $\mathsf{ftype}(\Gamma, T_0\{v/x\}, f) = T_f$; therefore, $\mathsf{ftype}(\Gamma, T_0', f) = T_f$.

  Thus, we can derive by F-GET, $\Gamma \vdash_{\mathsf{final}} p_0\{v/x\}.f : T_f\{p_0\{v/x\}/\texttt{this}\}$, which can be rewritten: $\Gamma \vdash_{\mathsf{final}} p_0.f\{v/x\} : (T_f\{p_0/\texttt{this}\})\{v/x\}$.

□

We write $\vdash S_1 \sqsubset^* S_2$ if $\mathsf{inh}(S_2) \subseteq \mathsf{inh}(S_1)$.

LEMMA B.4. *If* $x\!:\!T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v\!:\!T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash_{\mathsf{final}} p\!:\!T$, *and* $\Gamma \vdash T \trianglelefteq S$, *then* $\Gamma\{v/x\} \vdash_{\mathsf{final}} p\{v/x\}\!:\!T'$ *and* $\Gamma\{v/x\} \vdash T' \trianglelefteq S'$ *and* $S' \sqsubset^* S$.

PROOF: Follows from Lemma B.3 and Lemma B.29. □

LEMMA B.5. *If* $x\!:\!T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v\!:\!T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash T \trianglelefteq S$, *then* $\Gamma\{v/x\} \vdash T\{v/x\} \trianglelefteq S'$ *where* $\vdash S' \sqsubset^* S$.

PROOF: Follows from Lemma B.31 and Lemma B.2. □

LEMMA B.6. *If* $x\!:\!T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v\!:\!T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\mathsf{ftype}(\Gamma, T, f) = [\texttt{final}]\ T_f$, *then* $\mathsf{ftype}(\Gamma\{v/x\}, T\{v/x\}, f) = [\texttt{final}]\ T_f$.

Proof: Follows from Lemma B.5 and the definition of fields. □

Lemma B.7. *If* $x{:}T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v{:}T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\mathsf{mtype}(\Gamma, T, m) = (\overline{x}{:}\overline{T}) \to T_{n+1}$, *then* $\mathsf{mtype}(\Gamma\{v/x\}, T\{v/x\}, m) = (\overline{x}{:}\overline{T}) \to T_{n+1}$.

Proof: Follows from Lemma B.5 and the definition of methods. □

Lemma B.8. *If* $x{:}T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v{:}T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\mathsf{exact}(T)$, *then* $\mathsf{exact}(T\{v/x\})$.

Proof: By inspection of definition of exact. □

Lemma B.9. *If* $x{:}T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v{:}T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash T$ *then* $\Gamma\{v/x\} \vdash T\{v/x\}$.

Proof: Follows from Lemma B.2 and Lemma B.34. □

Lemma B.10. *If* $x{:}T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v{:}T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash T_1 \leq T_2$, *then* $\Gamma\{v/x\} \vdash T_1\{v/x\} \leq T_2\{v/x\}$.

Proof: By induction on the derivation of $\Gamma \vdash T_1 \leq T_2$.

- S-TRANS. Trivial via the induction hypothesis.
- S-SUP. Follows from the induction hypothesis and Lemma B.11. and S-TRANS.
- S-BOUND. By Lemma B.9 and Lemma B.5 and S-TRANS.
- S-NEST. Follows from the induction hypothesis and Lemma B.9.
- S-FIN. Lemma B.3.
- S-PRE-S1. Follows from the induction hypothesis and Lemma B.9.
- S-PRE-S2. By Lemma B.9.
- S-OUT. Follows from the induction hypothesis.
- S-IN. By Lemma B.9.
- S-MEET-LB. By Lemma B.9. Trivial.
- S-MEET-G. Follows from the induction hypothesis.
- S-ALIAS. Follows from definition of $\Gamma\{v/x\}$.
- S-EVAL. Trivial since $U_i\{v/x\} = U_i$.
- S-EXACT. Follows from the induction hypothesis and Lemma B.8.

□

Lemma B.11. *If* $x : T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v : T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $T\{\!\{\Gamma;\ T_y/y\}\!\} = T'$, *and* x *is not free in* T, *then* $T\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = T'\{v/x\}$.

Proof: The proof is by induction on type substitution derivation.

- $T = \circ$. Trivial.
- $T = T_0.C$. Follows from the induction hypothesis.
- $T = p.\texttt{class}$.
  - $p = v$. Trivial.
  - $p = z$. Trivial.
  - $p = y$. Then $T' = T_y$. and $T'\{v/x\} = T_y\{v/x\}$. By the definition of type substitution, $y.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = T_y\{v/x\}$. Done.
  - $p = x$. Vacuous (variable capture).
  - $p = p_0.f$.
    Let $p_0.\texttt{class}\{\!\{\Gamma;\ T_y/y\}\!\} = T_p$. and $p_0.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = T'_p$. By the induction hypothesis, $T'_p = T_p\{v/x\}$.
    - If $T_p = p_1.\texttt{class}$, then $T'_p = p_1\{v/x\}.\texttt{class}$. In this case, $p_0.f.\texttt{class}\{\!\{\Gamma;\ T_y/y\}\!\} = p_1.f.\texttt{class}$ and $p_0.f.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_y\{v/x\}/y\}\!\} = p_1\{v/x\}.f.\texttt{class} = p_1.f.\texttt{class}\{v/x\}$.
    - Otherwise, $T_p \neq p_1.\texttt{class}$. Since $\mathsf{ftype}(\Gamma, T_p, f) = T_f$, by Lemma B.6, $\mathsf{ftype}(\Gamma\{v/x\}, T_p\{v/x\}, f) = T_f$.
      Since by F-OK, $\emptyset \vdash T_f$, x is not free in T_f, and hence $T_f\{v/x\} = T_f$.

- $T = P\texttt{[}T_0\texttt{]}$. Follows from the induction hypothesis.
- $T = \texttt{\&}\overline{T}$. Follows from the induction hypothesis.

□

LEMMA B.12. (Substitution) *If* $x : T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v : T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\Gamma \vdash e : T$, *then* $\Gamma\{v/x\} \vdash e\{v/x\} : T\{v/x\}$.

PROOF: By induction on the derivation of $\Gamma \vdash e \vdash T$.
Let $e' = e\{v/x\}$ and $T' = T\{v/x\}$.

- T-FIN. Then $e = p$ and $T = p.\texttt{class}$ and $e' = p\{v/x\}$ and $T' = p\{v/x\}.\texttt{class}$. The case follows from Lemma B.3.
- T-GET. Then $e = e_0.f$ and $\Gamma \vdash e_0 : T_0$ and $\mathsf{ftype}(\Gamma, T_0, f) = [\texttt{final}]\ T_f$ and $T = T_f$.
  Then $e' = e_0\{v/x\}.f = e'_0.f$. By the induction hypothesis, since $\Gamma \vdash e_0 : T_0$, we have $\Gamma\{v/x\} \vdash e_0\{v/x\} : T_0\{v/x\}$.
  By Lemma B.6, and $\mathsf{ftype}(\Gamma\{v/x\}, T_0\{v/x\}, f) = [\texttt{final}]\ T_f$.
  Since by F-OK, $\emptyset \vdash T_f$, x is not free in T_f, and hence $T' = T_f\{v/x\} = T_f = T$.
  The case holds by T-GET.
- T-SET. The proof of this case is similar to the proof of the previous case for T-GET.
- T-SEQ. Follows from the induction hypothesis.
- T-NEW.
  Follows from Lemma B.9, Lemma B.6, and the induction hypothesis.
- T-CALL.
  Follows from IH, Lemma B.7, Lemma B.11, and Lemma B.8.
- T-SUB.
  Then $\Gamma \vdash e : T''$ where $\Gamma \vdash T'' \leq T$.
  By the induction hypothesis, $\Gamma\{v/x\} \vdash e\{v/x\} : T''\{v/x\}$.
  By Lemma B.10, $\Gamma\{v/x\} \vdash T''\{v/x\} \leq T\{v/x\}$.
  Thus, by T-SUB, $\Gamma\{v/x\} \vdash e\{v/x\} : T\{v/x\}$.

□

This lemma states that if a final access path p has a given type T, that type must be a supertype of $p.\texttt{class}$.

LEMMA B.13. *If* $\Gamma \vdash_{\mathsf{final}} p : T_p$ *and* $\Gamma \vdash p : T$, *then* $\Gamma \vdash p.\texttt{class} \leq T$.

PROOF: The proof is by induction on the height of the subtyping derivation.
There are only two ways to derive $\Gamma \vdash p : T$:

- T-FIN. Then $T = p.\texttt{class}$ and the case holds by A-REFL and S-ALIAS.
- T-SUB. Then $\Gamma \vdash p : T'$ and $\Gamma \vdash T' \leq T$.
  By the induction hypothesis, $\Gamma \vdash p.\texttt{class} \leq T'$. Therefore by S-TRANS, $\Gamma \vdash p.\texttt{class} \leq T$.

□

B.2 Type substitution again

This lemma states that a value substitution of v for x in a type results in a subtype of the type substitution of v's static type for x.

LEMMA B.14. (Type substitution) *If* $x : T_x \in \Gamma$, *and* $\Gamma\{v/x\} \vdash v : T_v$ *and* $\Gamma\{v/x\} \vdash T_v \leq T_x$, *and* $\vdash \Gamma\ \mathsf{ok}$, *and* $T\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T'$, *then* $\Gamma\{v/x\} \vdash T\{v/x\} \leq T'$.

PROOF: By induction on type substitution derivation.

- Case $T = \circ$. Trivial.
- Case $T = T_0.C$. Then $T\{v/x\} = T_0\{v/x\}.C$ and $T' = T'_0.C$ where $T_0\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T'_0$.
  By the induction hypothesis $\Gamma\{v/x\} \vdash T_0\{v/x\} \leq T'_0$; therefore, by S-NEST, $\Gamma\{v/x\} \vdash T_0\{v/x\}.C \leq T'_0.C$.
- Case $T = p.\texttt{class}$. Then $T\{v/x\} = p\{v/x\}.\texttt{class}$.
  - Case $p = v$. Trivial.
  - Case $p = y \neq x$. Trivial.
  - Case $p = x$. Then $T = x.\texttt{class}$ and $T' = T_v$ and $T\{v/x\} = v.\texttt{class}$.
    Since $\Gamma\{v/x\} \vdash v : T'$, $\Gamma\{v/x\} \vdash v.\texttt{class} \leq T'$ by Lemma B.13.

  - Case $p = p_0.f$. Then $T = p_0.f.\texttt{class}$ and $T\{v/x\} = p_0\{v/x\}.f.\texttt{class}$
    Let $p_0.\texttt{class}\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T_p$. By the induction hypothesis, $\Gamma\{v/x\} \vdash p_0\{v/x\}.\texttt{class} \leq T_p$.
    There are two cases.
    - $T_p \neq p_0'.\texttt{class}$ for any p_0'.
      Then, $\mathsf{ftype}(\Gamma\{v/x\}, T_p, f) = T_f$.
      By F-GET, we have $\Gamma\{v/x\} \vdash_{\mathsf{final}} p_0\{v/x\}.f : T_f$.
      Thus, by Lemma B.13, $\Gamma\{v/x\} \vdash p_0\{v/x\}.f.\texttt{class} \leq T_f\{p_0\{v/x\}/\texttt{this}\}$. Since T_f has no free variables, $T_f = T_f\{p_0\{v/x\}/\texttt{this}\}$.
    - $T_p = p_0'.\texttt{class}$. It must be that $p_0' = p_0\{v/x\}$. The case follows trivially from S-ALIAS.

- Case $T = P[T_0]$. Then $T\{v/x\} = P[T_0\{v/x\}]$. and $T' = P[T_0']$ where and $T_0\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T_0'$.
  By the induction hypothesis, $\Gamma\{v/x\} \vdash T_0\{v/x\} \leq T_0'$; therefore, by S-PRE-S1, $\Gamma\{v/x\} \vdash T\{v/x\} \leq T'$.
- Case $T = \&\overline{T}$. Then $T\{v/x\} = \&\overline{T\{v/x\}}$ and $T' = \&\overline{T'}$ where for all i, $T_i\{\!\{\Gamma\{v/x\};\ T_v/x\}\!\} = T_i'$. By the induction hypothesis, $\Gamma\{v/x\} \vdash T_i\{v/x\} \leq T_i'$; therefore, by S-MEET-G, $\Gamma\{v/x\} \vdash T\{v/x\} \leq T'$.

□

B.3 Environments

We say a heap H_2 remaps H_1 if

- $H_1 = \emptyset$ and $H_2 = \emptyset$, or
- H_2' remaps H_1', and $H_1 = H_1', \ell \mapsto S\ \{\overline{f} = \overline{v}\}$, and $H_2 = H_2', \ell \mapsto S\ \{\overline{f} = \overline{v'}\}$, and for all f_i, if $\mathsf{ftype}(\emptyset, S, f_i) = \texttt{final}\ T$, then $v_i = v_i'$.

H_2 extends H_1 if H_2 remaps H_1, or there is an H such that H extends H_1 and $H_2 = H, \ell \mapsto o$ and $\ell \notin \mathrm{dom}(H)$.
We say an environment Γ_2 extends Γ_1 if there is a Γ such that $\Gamma_2 = \Gamma_1, \Gamma$.

LEMMA B.15. *If H_2 remaps H_1, then $H_2^\dagger$ extends $H_1^\dagger$.*

PROOF: By structural induction on H_2.

- $H_2 = \emptyset$. Then $H_1 = \emptyset$. Trivial.
- $H_2 = H_2', \ell \mapsto S\ \{\overline{f} = \overline{v}\}$. Then $H_2 = H_1', \ell \mapsto S\ \{\overline{f} = \overline{v'}\}$. By the induction hypothesis, $H_2'^\dagger = H_1'^\dagger$. For all final fields f of S, $o_1[f] = o_2[f]$. Therefore, $H_2^\dagger = H_1^\dagger$ by construction.

□

LEMMA B.16. *If H_2 extends H_1, then $H_2^\dagger$ extends $H_1^\dagger$.*

PROOF: By structural induction on H.
If H_2 remaps H_1, then $H_2^\dagger = H_1^\dagger$ by Lemma B.15.
Otherwise, there is an H such that H extends H_1 and $H_2 = H, \ell \mapsto S\ \{\overline{f} = \overline{v}\}$ and $\ell \notin \mathrm{dom}(H)$.
By the induction hypothesis, $H^\dagger$ extends $H_1^\dagger$ and by construction $H_2^\dagger = H^\dagger, \ell : S, \ell.\overline{f'} = \overline{v'}$ where $\overline{f'}$ are the final fields of S. □

LEMMA B.17. (Extension) *If $\Gamma \vdash e : T$ and $\vdash \Gamma, \Gamma'$ ok, then $\Gamma, \Gamma' \vdash e : T$.*

PROOF: By induction on the derivation of $\Gamma \vdash e : T$. □

LEMMA B.18. *If $\Gamma, x : T_v, \Gamma' \vdash e : T$ or $\Gamma, x.\texttt{class} = T_v, \Gamma' \vdash e : T$, and Γ' contains no $y : T_y$, then $\Gamma, \Gamma', x : T_v \vdash e : T$.*

PROOF: By induction on the derivation of $\Gamma, x : T_v, \Gamma' \vdash e : T$ and by induction on the derivation of $\Gamma, x.\texttt{class} = T_v, \Gamma' \vdash e : T$. □

LEMMA B.19. *If $\Gamma, x : T_v \vdash e : T$ or if $\Gamma, x.\texttt{class} = T_v \vdash e : T$ and x is not free in e, then $\Gamma \vdash e : T$.*

PROOF: By induction on the derivation of $\Gamma, x : T_v \vdash e : T$ and by induction on the derivation of $\Gamma, x.\texttt{class} = T_v \vdash e : T$. □

LEMMA B.20. *If $\Gamma \vdash T \trianglelefteq S$ and $\vdash \Gamma, \Gamma'$ ok, then $\Gamma, \Gamma' \vdash T \trianglelefteq S$.*

PROOF: By induction on the derivation of $\Gamma \vdash T \trianglelefteq S$. □

LEMMA B.21. *If $\Gamma \vdash T$ and $\vdash \Gamma, \Gamma'$ ok, then $\Gamma, \Gamma' \vdash T$.*

PROOF: By induction on the derivation of $\Gamma \vdash T$. □

LEMMA B.22. *If* $\Gamma \vdash T_1 \leq T_2$ *and* $\vdash \Gamma, \Gamma'$ ok*, then* $\Gamma, \Gamma' \vdash T_1 \leq T_2$.

PROOF: By induction on the derivation of $\Gamma \vdash T_1 \leq T_2$. □

LEMMA B.23. *If* $\Gamma \vdash_{\mathsf{final}} p : T$*, and* $\vdash \Gamma, \Gamma'$ ok*, then* $\Gamma, \Gamma' \vdash_{\mathsf{final}} p : T$.

PROOF: By induction on the derivation of $\Gamma \vdash_{\mathsf{final}} p : T$. □

LEMMA B.24. *If* $\mathsf{ftype}(\Gamma, T, f) = T_f$ *and* $\vdash \Gamma, \Gamma'$ ok*, then* $\mathsf{ftype}((\Gamma, \Gamma'), T, f) = T_f$

PROOF: By structural induction on Γ. □

LEMMA B.25. *If* $\mathsf{mtype}(\Gamma, T, m) = (\overline{x} : \overline{T}) \rightarrow T_{n+1}$*, and* $\vdash \Gamma, \Gamma'$ ok*, then* $\mathsf{mtype}((\Gamma, \Gamma'), T, m) = (\overline{x} : \overline{T}) \rightarrow T_{n+1}$,

PROOF: By structural induction on Γ. □

LEMMA B.26. *If* $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$ *and* $\vdash \Gamma, \Gamma'$ ok*, then If* $T\{\!\{\Gamma, \Gamma';\ T_v/x\}\!\} = T'$

PROOF: By induction on the type substitution derivation.

- $T = \circ$. Then $T' = \circ$. Trivial.
- $T = T_0.C$. Follows from the induction hypothesis.
- $T = p.\mathtt{class}$.
  - $p = v$. Trivial since the environment is not used.
  - $p = y$. Trivial since the environment is not used.
  - $p = p_0.f.\mathtt{class}$ This is only case where the environment is used. If $p_0.\mathtt{class}\{\!\{\Gamma;\ T_v/x\}\!\} = T_p$, then by the induction hypothesis, $p_0.\mathtt{class}\{\!\{\Gamma;\ T_v/x\}\!\} = T_p$. If T_p is not a path type, the case holds by Lemma B.24. Otherwise, the case holds trivially.
- $T = P[T_0]$. Follows from the induction hypothesis.
- $T = \&\overline{T}$. Follows from the induction hypothesis.

□

LEMMA B.27. *If* $\Gamma \vdash T_1 \simeq T_2$ *and* $\Gamma \vdash T_1 \trianglelefteq S_1$ *and* $\Gamma \vdash T_2 \trianglelefteq S_2$*, then* $\Gamma \vdash S_1 \approx S_2$ *(XXX).*

PROOF: XXX □

LEMMA B.28. *If* $\Gamma \vdash T_1 \leq T_2$*, and* $\mathsf{exact}(T_2)$*, and* $\Gamma \vdash T_1 \trianglelefteq S_1$ *and* $\Gamma \vdash T_2 \trianglelefteq S_2$*, then* $\Gamma \vdash S_1 \approx S_2$ *(XXX).*

PROOF: XXX □

LEMMA B.29. *If* $\Gamma \vdash T_1 \leq T_2$ *and* $\Gamma \vdash T_1 \trianglelefteq S_1$ *and* $\Gamma \vdash T_2 \trianglelefteq S_2$*, then* $\vdash S_1 \sqsubset^* S_2$.

PROOF: By induction on the subtyping derivation.

- S-TRANS. Trivial via the induction hypothesis.
- S-SUP. Follows from Lemma B.31.
- S-BOUND. Trivial since $T_2 = S_1 = S_2$.
- S-NEST. Follows from the induction hypothesis.
- S-FIN. Follows from BD-FIN.
- S-PRE-S1. Follows from definition of prefix and BD-PRE.
- S-PRE-S2. Follows from definition of prefix and BD-PRE.
- S-OUT. Follows from the induction hypothesis.
- S-IN. Follows from definition of prefix and BD-PRE.
- S-MEET-LB. Follows from BD-MEET.
- S-MEET-G. Follows from the induction hypothesis.

- S-ALIAS. Follows from Lemma B.27.
- S-EVAL. Trivial since $S_1 = S_2$.
- S-EXACT. Follows from Lemma B.28.

□

LEMMA B.30. *and* $\Gamma \vdash T_1 \leq T_2$, *and* $\mathsf{ftype}(\Gamma, T_1, f) = [\texttt{final}]\ T_f$, *then* $\mathsf{ftype}(\Gamma, T_2, f) = [\texttt{final}]\ T_f$.

PROOF: Let $\Gamma \vdash T_1 \trianglelefteq S_1$ and $\Gamma \vdash T_2 \trianglelefteq S_2$. By Lemma B.29, $\vdash S_1 \sqsubset^* S_2$. By the definition of fields Thus, $\mathsf{fields}(\Gamma, S') \supseteq \mathsf{fields}(\Gamma, S)$. Therefore, for all fields f of S, $\mathsf{ftype}(\Gamma, T', f) = \mathsf{ftype}(\Gamma, T, f)$. □

B.4 Type substitution redux

LEMMA B.31. *If* $x : T_x \in \Gamma$, *and* $\Gamma \vdash T_v \leq T_x$, *and* $\Gamma \vdash T \trianglelefteq S$, *and* $\Gamma \vdash T_v$, *and* $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$, *then* $\Gamma \vdash T' \trianglelefteq S'$ *and* $S' \sqsubset^* S$.

PROOF: By induction on type substitution derivation.

- $T = \circ$. Trivial since $T' = T$.
- $T = T_0.C$. Then $T' = T_0'.C$ where and $T_0\{\!\{\Gamma;\ T_v/x\}\!\} = T_0'$.

  Let $\Gamma \vdash T_0 \trianglelefteq S_0$. By the induction hypothesis, $\Gamma \vdash T_0' \trianglelefteq S_0'$ and $\vdash S_0' \sqsubset^* S_0$. By BD-NEST, $\Gamma \vdash T_0'.C \trianglelefteq S_0'.C$. The case holds by the definitions of mem and inh [XXX cleanup].
- $T = p.\texttt{class}$.

  Then by BD-FIN, $\Gamma \vdash_{\mathsf{final}} p : T_0$. and $\Gamma \vdash T_0 \trianglelefteq S$.
  - $p = v$. Then $T' = T$.
  - $p = y$. Then $T' = T$.
  - $p = x$. Then $T' = T_v$. and $\Gamma \vdash T_v \trianglelefteq S'$.

    Since $T = x.\texttt{class}$,

    Since $x : T_x \in \Gamma$, $\Gamma \vdash_{\mathsf{final}} x : T_x$ by F-VAR.

    Since $\Gamma \vdash x.\texttt{class} : S$, by BD-FIN we have $\Gamma \vdash T_x \trianglelefteq S$.

    Since $\Gamma \vdash T_v \leq T_x$, by Lemma B.29, $\vdash S' \sqsubset^* S$.
  - $p = p_0.f$. Let $p_0.\texttt{class}\{\!\{\Gamma;\ T_v/x\}\!\} = T_p$. By F-GET, $\Gamma \vdash_{\mathsf{final}} p_0 : T_p$, $\mathsf{ftype}(\Gamma, T_p, f) = \texttt{final}\ T_f$, and $\Gamma \vdash_{\mathsf{final}} p_0.f : T_f$. By BD-FIN, we have $\Gamma \vdash T_f \trianglelefteq S$.
    - If $T_p = p_0'.\texttt{class}$, then $T' = p_0'.f.\texttt{class}$. By F-GET, $\Gamma \vdash_{\mathsf{final}} p_0' : T_p'$, $\mathsf{ftype}(\Gamma, T_p', f) = \texttt{final}\ T_f$, and $\Gamma \vdash_{\mathsf{final}} p_0'.f : T_f$. Since $\Gamma \vdash_{\mathsf{final}} T_f \trianglelefteq S$, by BD-FIN, $\Gamma \vdash_{\mathsf{final}} T' \trianglelefteq S$ and $S = S'$.
    - Otherwise, T_p is not a path type. Then by the definition of type substitution, $T' = T_f$ where $\mathsf{ftype}(\Gamma, T_p, f) = T_f$. Since $\Gamma \vdash_{\mathsf{final}} T_f \trianglelefteq S$, we have $\Gamma \vdash_{\mathsf{final}} T' \trianglelefteq S$ and $S = S'$.
- $T = P[T_0]$. Let $\Gamma \vdash T_0 \trianglelefteq S_0$. By the induction hypothesis, $\Gamma \vdash T_0' \trianglelefteq S_0'$ and $\vdash S_0' \sqsubset^* S_0$. Thus, by BD-PRE, $\Gamma \vdash P[T_0'] \trianglelefteq S'$ where $S' = \mathsf{prefix}(P, S_0')$. Since $S = \mathsf{prefix}(P, S_0)$, by the definition of prefix, $\vdash S' \sqsubset^* S$ [XXX cleanup].
- $T = \&\overline{T}$. Then $T' = \&\overline{T'}$. Let $\Gamma \vdash T_i \trianglelefteq S_i$. By the induction hypothesis, for all i $\Gamma \vdash T_i' \trianglelefteq S_i'$ and $\vdash S_i' \sqsubset^* S_i$. By BD-MEET, $\Gamma \vdash T_i' \trianglelefteq S_i'$. The case holds by the definitions of mem and inh [XXX cleanup].

  □

LEMMA B.32. *If* $\vdash S_1 \sqsubset^* S_0$, *then* $\vdash \mathsf{prefix}(P, S_1) \sqsubset^* \mathsf{prefix}(P, S_0)$.

PROOF: Then $\mathsf{inh}(S_1) \supseteq \mathsf{inh}(S_0)$.

Thus by the definition of prefix, $\mathsf{mem}(\mathsf{prefix}(P, S_1)) \supseteq \mathsf{mem}(\mathsf{prefix}(P, S_0))$.

Hence, $\mathsf{inh}(\mathsf{prefix}(P, S_1)) \supseteq \mathsf{inh}(\mathsf{prefix}(P, S_0))$, and thus $\vdash \mathsf{prefix}(P, S_1) \sqsubset^* \mathsf{prefix}(P, S_0)$.

□

LEMMA B.33. *If* $x : T_x \in \Gamma$, *and* $\Gamma \vdash T_v \leq T_x$, *and* $\mathsf{ftype}(\Gamma, T, f) = [\texttt{final}]\ T_f$, *and* $\Gamma \vdash T_v$, *and* $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$, *then* $\mathsf{ftype}(\Gamma, T', f) = [\texttt{final}]\ T_f$.

PROOF: Let $\Gamma \vdash T \trianglelefteq S$ and $\Gamma \vdash T' \trianglelefteq S'$. By Lemma B.31, and $\vdash S' \sqsubset^* S$. By the definition of fields Thus, $\mathsf{fields}(\Gamma, S') \supseteq \mathsf{fields}(\Gamma, S)$. Therefore, for all fields f of S, $\mathsf{ftype}(\Gamma, T', f) = \mathsf{ftype}(\Gamma, T, f)$. □

LEMMA B.34. *If* $x : T_x \in \Gamma$, *and* $\Gamma \vdash T_v \leq T_x$, *and* $\Gamma \vdash T$, *and* $\Gamma \vdash T_v$, *and* $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$, *then* $\Gamma \vdash T'$.

PROOF: By induction on type substitution derivation.

- $T = \circ$. Trivial.
- $T = T_0.C$. Then $T\{\!\{\Gamma;\ T_v/x\}\!\} = T_0'.C = T_0\{\!\{\Gamma;\ T_v/x\}\!\}.C$. By the induction hypothesis, T_0' is well-formed. Let $\Gamma \vdash T_0 \trianglelefteq S_0$ and $\Gamma \vdash T_0' \trianglelefteq S_0'$. By Lemma B.31, $\vdash S_0' \sqsubset^* S_0$. Therefore $\vdash S_0'.C$ defined, and by WF-NEST, $\Gamma \vdash T_0'.C$.
- $T = p.\texttt{class}$.

  By cases on p.

  - $p = v$. Trivial.
  - $p = y \neq x$. Trivial.
  - $p = x$. Then $T\{\!\{\Gamma;\ T_v/x\}\!\} = T_v$ and the case follows from the assumptions.
  - $p = p_0.f$. Let $p_0.\texttt{class}\{\!\{\Gamma;\ T_v/x\}\!\} = T_p$.

    By Lemma B.33, $\mathsf{ftype}(\Gamma, T_p, f) = \mathsf{ftype}(\Gamma, p_0.\texttt{class}, f) = T_f$.

    There are two cases:

    - Assume $T_p \neq p_0'.\texttt{class}$. Then $T' = T_f$. By the induction hypothesis, $\Gamma \vdash T_f$.
    - If $T_p = p_0'.\texttt{class}$, then $T' = p_0'.f.\texttt{class}$. Since by Lemma B.33, ftype is unchanged by the substitution, we can derive $\Gamma \vdash_{\mathsf{final}} p_0'.f : T_f$ by F-GET. Hence, by WF-FIN, we have $\Gamma \vdash p_0'.f.\texttt{class}$.
- $T = P\,\texttt{[}T_0\texttt{]}$. Then $T' = P\,\texttt{[}T_0'\texttt{]}$ where $T_0\{\!\{\Gamma;\ T_v/x\}\!\} = T_0'$.

  By the induction hypothesis, T_0' is well-formed. Let $\Gamma \vdash T_0 \trianglelefteq S_0$ and $\Gamma \vdash T_0' \trianglelefteq S_1$.

  By Lemma B.31, $\vdash S_1 \sqsubset^* S_0$.

  Therefore, by Lemma B.32, $\vdash \mathsf{prefix}(P, S_1) \sqsubset^* \mathsf{prefix}(P, S_0)$. Hence, $\mathsf{prefix}(P, S_1) \neq \texttt{\&nil}$.

  Finally, by WF-PRE, $\Gamma \vdash T'$
- $T = \texttt{\&}\overline{T}$. Then $T' = \texttt{\&}\overline{T'}$ where for all i $T_i\{\!\{\Gamma;\ T_v/x\}\!\} = T_i'$.

  By the induction hypothesis, T_i' is well-formed. Since by Lemma B.35, $|\mathsf{exacts}(T)| \leq |\mathsf{exacts}(T')|$, and since $\neg\mathsf{exact}(T)$ implies $\neg\mathsf{exact}(T')$, we can derive $\Gamma \vdash T'$ by WF-MEET.

□

LEMMA B.35. *If $x : T_x \in \Gamma$, and $\Gamma \vdash T_v \leq T_x$, and $\Gamma \vdash T_v$, and $T\{\!\{\Gamma;\ T_v/x\}\!\} = T'$, then $|\mathsf{exacts}(T)| \leq |\mathsf{exacts}(T')|$.*

PROOF: By inspection of type substitution rules, no rule substitutes an exact type for an inexact type, and since T_v is well-formed, by Lemma B.36, $|\mathsf{exacts}(T_v)| \leq 1$, so substituting T_v for an □

LEMMA B.36. *If $\Gamma \vdash T$, then $|\mathsf{exacts}(T)| \leq 1$.*

PROOF: By structural induction on T.

- $T = \circ$. Then $|\mathsf{exacts}(T)| = 0$.
- $T = T_0.C$. Then $|\mathsf{exacts}(T)| = |\mathsf{exacts}(T_0)|$.
- $T = p.\texttt{class}$. Then $|\mathsf{exacts}(T)| = 1$.
- $T = P\,\texttt{[}T_0\texttt{]}$. Then either $|\mathsf{exacts}(T)| = 1$, or $|\mathsf{exacts}(T)| = |\mathsf{exacts}(T_0)|$.
- $T = \texttt{\&}\overline{T}$. Then $|\mathsf{exacts}(T)| \leq 1$ by WF-MEET.

□

B.5 Non-dependent types redux

LEMMA B.37. *If $\vdash P_1 \sqsubset P_2$, then $\emptyset \vdash P_1 \leq P_2$.*

PROOF: By induction on the derivation of $\vdash P_1 \sqsubset P_2$.
There are two cases:

- If $\vdash P_1 \sqsubset_{\mathsf{sc}} P_2$, then $P_1 = P_1'.C$ and there is a P such that $\vdash P_1' \sqsubset^* P$, and $CT(P.C) = \texttt{class}\ C\ \texttt{extends}\ T\ \{\ldots\}$, and $T\{\!\{\emptyset;\ P_1'/\texttt{this}\}\!\} = S$, and $P_2 \in \mathsf{mem}(S)$.

  By the induction hypothesis and S-TRANS, $\emptyset \vdash P_1' \leq P$. Thus, by S-SUP, we can derive $\emptyset \vdash P_1'.C \leq P_2$.
- If $\vdash P_1 \sqsubset_{\mathsf{fb}} P_2$, then $P_1 = P_1'.C$ and $P_2 = P_2'.C$ and $\vdash P_1' \sqsubset P_2'$. By the induction hypothesis, $\emptyset \vdash P_1' \leq P_2'$. By S-NEST, $\emptyset \vdash P_1 \leq P_2$.

□

LEMMA B.38. *If $P \in \mathsf{inh}(S)$, then $\emptyset \vdash S \leq P$.*

PROOF: Trivial from Lemma B.37. □

LEMMA B.39. *If $\Gamma \vdash p : T$ and $\Gamma \vdash_{\mathsf{final}} p : T_p$, then $\Gamma \vdash T_p \leq T$.*

PROOF: By induction on the derivation of $\Gamma \vdash p : T$.
Only three cases apply:

- T-FIN.
  Since $\Gamma \vdash_{\mathsf{final}} p : T_p$, by T-FIN $\Gamma \vdash p : p.\mathtt{class}$, and by S-FIN $\Gamma \vdash p.\mathtt{class} \leq T_p$.
- T-GET.
  Then $p = p_0.f$, and $\Gamma \vdash p_0 : T_0$, and $\mathsf{ftype}(\Gamma, T_0, f) = \mathtt{final}\ T$.
  Since $\Gamma \vdash_{\mathsf{final}} p_0.f : T_p$, by F-GET we have $\Gamma \vdash_{\mathsf{final}} p_0 : T_1$ and $\mathsf{ftype}(\Gamma, T_1, f) = \mathtt{final}\ T_p$.
  Thus, $T = T_p$.
- T-SUB.
  Then $\Gamma \vdash p : T'$ and $\Gamma \vdash T' \leq T$.
  By the induction hypothesis, and $\Gamma \vdash T_p \leq T'$.
  Hence, by S-TRANS, $\Gamma \vdash T_p \leq T$.

  □

LEMMA B.40. *If $\mathsf{mtype}(\emptyset, S, m) = (\overline{x} : \overline{T}) \to T_{n+1}$, then $\mathsf{mbody}(S, m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e\}$.*

PROOF: Follows immediately from definition of mtype and mbody. □

B.6 Subject reduction

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

LEMMA B.41. *If $\vdash p, H$, and $H^\dagger \vdash_{\mathsf{final}} p : T$, and $p, H \longrightarrow p', H$, and $H^\dagger \vdash_{\mathsf{final}} p' : T'$, then $H^\dagger \vdash p = p'$.*

PROOF: By induction on the derivation of $H^\dagger \vdash_{\mathsf{final}} p : T$.
Since p can make a step, $p = p_0.f$. We consider p_0 by cases.

- $p_0 = \mathtt{null}$. Then $p = \mathtt{null}.f$ and R-NULL is the only rule that can apply.
- $p_0 = \ell$. Then $p = \ell.f$ and R-GET is the only rule that can apply, $p' = v_i = H(\ell)[f_i]$ where $H(\ell) = S\ \{\overline{f} = \overline{v}\}$.
  By the construction of $H^\dagger$, $H^\dagger$ must include $\ell.f_i = v_i$.
- $p_0 \neq v$.
  Then R-CONG is the only rule that can apply and $p_0, H \longrightarrow p'_0, H$.
  By F-GET, $H^\dagger \vdash_{\mathsf{final}} p_0 : T_0$.
  By the induction hypothesis, $H^\dagger \vdash p_0 = p'_0$.
  Thus, by A-FIELD, $H^\dagger \vdash p_0.f = p'_0.f$.

  □

LEMMA B.42. *If $\vdash p, H$, and $H^\dagger \vdash_{\mathsf{final}} p : T$, and $p, H \longrightarrow p', H$, then $\vdash p', H$ and $H^\dagger \vdash_{\mathsf{final}} p' : T'$, where $H^\dagger \vdash T' \leq T$.*

PROOF: By induction on the derivation of $H^\dagger \vdash_{\mathsf{final}} p : T$.
Since p can make a step, $p = p_0.f$. We consider p_0 by cases.

- $p_0 = \mathtt{null}$. Then $p = \mathtt{null}.f$ and R-NULL is the only rule that can apply.
- $p_0 = \ell$. Then $p = \ell.f$ and R-GET is the only rule that can apply, $p' = v_i = H(\ell)[f_i]$ where $H(\ell) = S\ \{\overline{f} = \overline{v}\}$.
  By F-GET, $H^\dagger \vdash_{\mathsf{final}} \ell ty T_0$, and $\mathsf{ftype}(H^\dagger, T_0, f) = T$.
  Since $\vdash p, H$, by CONFIG and HEAP, we have $H \vdash \ell$. Thus, by H-LOC, we have $H^\dagger \vdash v_i \vdash T$.
  By Lemma B.39, $H^\dagger \vdash_{\mathsf{final}} p' : T'$, where $H^\dagger \vdash T' \leq T$.
  By H-LOC, we can also derive $v_i \in \mathrm{dom}(H) \cup \{\mathtt{null}\}$.
  If $v_i = \ell_i$, then $v_i \in \mathrm{dom}(H)$. Therefore by CONFIG, $\vdash v_i, H$.

- $p_0 \neq v$.
  Then R-CONG is the only rule that can apply and $p_0, H \longrightarrow p_0', H$.
  By F-GET, $H^\dagger \vdash_{\mathsf{final}} p_0 : T_0$, and $\mathsf{ftype}(H^\dagger, T_0, f) = T$.
  By the induction hypothesis, $H^\dagger \vdash_{\mathsf{final}} p_0' : T_0'$ and $H^\dagger \vdash T_0' \leq T_0$.
  By Lemma B.30, $\mathsf{ftype}(H^\dagger, T_0', f) = T$.
  Hence, we can derive by F-GET, $H^\dagger \vdash_{\mathsf{final}} p_0'.f : T$.
  By the induction hypothesis, $\vdash p_0', H$. Therefore, since $\mathsf{locs}(p_0'.f) = \mathsf{locs}(p_0')$, by CONFIG we can derive $\vdash p_0'.f, H$.

  □

LEMMA B.43. *If $H^\dagger \vdash TE[p] \trianglelefteq S$ and $p, H \longrightarrow p', H$, then $H^\dagger \vdash TE[p'] \trianglelefteq S'$ where $\vdash S' \sqsubset^* S$.*

PROOF: By induction on $H^\dagger \vdash TE[p] \trianglelefteq S$.

- $TE = TE_0.C$. Then $TE[p] = TE_0[p].C$.
  By BD-NEST, $H^\dagger \vdash TE_0[p] \trianglelefteq S_0$ where $S = S_0.C$.
  By the induction hypothesis, $H^\dagger \vdash TE_0[p'] \trianglelefteq S_0'$.
  Thus, we can derive by BD-NEST. $H^\dagger \vdash TE_0[p'] \trianglelefteq S_0'.C$.
  Also, by the induction hypothesis, $\vdash S_0' \sqsubset^* S_0$.
  Since $\vdash S_0' \sqsubset^* S_0$, we have $\vdash S_0'.C \sqsubset^* S_0.C$.
- $TE = E.\texttt{class}$. Then $TE[p] = E.\texttt{class}[p]$.
  By BD-FIN, $H^\dagger \vdash_{\mathsf{final}} E.\texttt{class}[p] : T$ and $H^\dagger \vdash T : S$.
  By Lemma B.42, $H^\dagger \vdash_{\mathsf{final}} E.\texttt{class}[p'] : T'$ where $H^\dagger \vdash T' \leq T$.
  Let $H^\dagger \vdash T' \trianglelefteq S'$. By BD-FIN, we can derive $H^\dagger \vdash E.\texttt{class}[p'] \trianglelefteq S'$.
  And, by Lemma B.29, $\vdash S' \sqsubset^* S$
- $TE = P\,\texttt{[}TE_0\texttt{]}$. Then $TE[p] = P\,\texttt{[}TE_0[p]\texttt{]}$.
  By BD-PRE, $H^\dagger \vdash TE_0[p] : S_0$, and $(\,\texttt{[}P\texttt{]}\,, S_0) = S$.
  By the induction hypothesis, $H^\dagger \vdash TE_0[p'] : S_0'$ where $\vdash S_0' \sqsubset^* S_0$.
  By Lemma B.32, $S' = (\,\texttt{[}P\texttt{]}\,, S_0') \sqsubset^* (\,\texttt{[}P\texttt{]}\,, S_0) = S$.
  Thus, by BD-PRE, $H^\dagger \vdash P\,\texttt{[}TE_0[p']\texttt{]} \trianglelefteq S'$.
- $TE = \&(\overline{U}, TE_0, \overline{T})$. Then $TE[p] = \&(\overline{U}, TE_0[p], \overline{T})$.
  By BD-MEET, $H^\dagger \vdash TE_0[p]$.
  By the induction hypothesis, $H^\dagger \vdash TE_0[p']$.
  All other components of the intersection do not change and therefore remain well-formed.
  Thus, we can derive by BD-MEET, $H^\dagger \vdash \&(\overline{U}, TE_0[p'], \overline{T})$.

  □

LEMMA B.44. *If $H^\dagger \vdash_{\mathsf{final}} p$ and $p, H \longrightarrow p', H$, then $H^\dagger \vdash p = p'$.*

PROOF: By induction on the structure of p.

- $p = v$. Vacuous since p cannot take a step.
- $p = x$. Vacuous since p is not well-formed.
- $p = \ell.f_i$. Then $H(\ell) = S\ \{\overline{f} = \overline{v}\}$ and $p' = v_i$. By the definition of $H^\dagger$, we have $\ell.f_i = v_i \in H^\dagger$. Therefore, by A-ENV, $H^\dagger \vdash \ell.f_i = v_i$.
- $p = p_0.f_i$ where p_0 is not a location ℓ. Then $p_0, H \longrightarrow p_0', H$. By the induction hypothesis, $H^\dagger \vdash p_0 = p_0'$. Thus, by A-FIELD, we have $H^\dagger \vdash p_0.f = p_0'.f$.

  □

LEMMA B.45. *If $H^\dagger \vdash TE[p]$ and $p, H \longrightarrow p', H$, then $H^\dagger \vdash TE[p']$.*

PROOF: By induction on $H^\dagger \vdash TE[p]$.

- $TE = TE_0.C$. Then $TE[p] = TE_0[p].C$.

  By WF-NEST, $H^\dagger \vdash TE_0[p]$, $H^\dagger \vdash TE_0[p] \trianglelefteq S$, and $\vdash S.C$ defined.

  By the induction hypothesis, $H^\dagger \vdash TE_0[p']$.

  By Lemma B.43, $H^\dagger \vdash TE_0[p'] \trianglelefteq S'$ where $\vdash S' \sqsubset^* S$.

  Since $\vdash S' \sqsubset^* S$ and Since $\vdash S.C$ defined, $\vdash S'.C$ defined.

  Thus, we can derive by WF-NEST. $H^\dagger \vdash TE_0[p']$.
- $TE = E.\texttt{class}$. Then $TE[p] = E.\texttt{class}[p]$.

  By WF-FIN, then $H^\dagger \vdash_{\mathsf{final}} E.\texttt{class}[p] : T$. By Lemma B.42, $H^\dagger \vdash_{\mathsf{final}} E.\texttt{class}[p'] : T'$.

  Hence, by WF-FIN, we can derive $H^\dagger \vdash E.\texttt{class}[p']$.
- $TE = P\texttt{[}TE_0\texttt{]}$. Then $TE[p] = P\texttt{[}TE_0[p]\texttt{]}$.

  By WF-PRE, $H^\dagger \vdash P$, $H^\dagger \vdash TE_0[p]$, and $H^\dagger \vdash P\texttt{[}TE_0[p]\texttt{]} \trianglelefteq S$.

  By the induction hypothesis, $H^\dagger \vdash TE_0[p']$.

  By BD-PRE, $H^\dagger \vdash TE_0[p] \trianglelefteq S_0$. By Lemma B.43, $H^\dagger \vdash TE_0[p'] \trianglelefteq S_0'$, where $\vdash S_0' \sqsubset^* S_0$.

  By Lemma B.32, $S' = (\texttt{[}P\texttt{]}, S_0') \sqsubset^* (\texttt{[}P\texttt{]}, S_0) = S$. Thus, by BD-PRE, $H^\dagger \vdash P\texttt{[}TE_0[p']\texttt{]} \trianglelefteq S'$.

  Hence, by WF-PRE, we can derive $H^\dagger \vdash P\texttt{[}TE_0[p']\texttt{]}$.
- $TE = \texttt{\&}(\overline{U}, TE_0, \overline{T})$. Then $TE[p] = \texttt{\&}(\overline{U}, TE_0[p], \overline{T})$.

  By WF-MEET, $H^\dagger \vdash TE_0[p]$.

  By the induction hypothesis, $H^\dagger \vdash TE_0[p']$.

  All other components of the intersection do not change and therefore remain well-formed.

  Since the structure of $TE_0[p]$ and $TE_0[p']$ are the same, we have $\mathsf{prefixExact}_k(TE_0[p]) = \mathsf{prefixExact}_k(TE_0[p'])$.

  Since all T_i in $\mathsf{exacts}(TE[p])$ are equivalent up to aliasing, and since by Lemma B.44 $H^\dagger \vdash p = p'$, we have all T_i in $\mathsf{exacts}(TE[p'])$ are equivalent up to aliasing,

  Thus, we can derive by WF-MEET, $H^\dagger \vdash \texttt{\&}(\overline{U}, TE_0[p'], \overline{T})$.

  □

LEMMA B.46. *If* $\vdash E[e], H$, *and* $e, H \longrightarrow e', H$, *and* $\vdash e', H$, *then* $\vdash E[e'], H$.

PROOF: Since $\mathsf{locs}(E[e']) \subseteq \mathsf{locs}(E[e]) \cup \mathsf{locs}(e')$, and $\mathsf{locs}(E[e]) \subseteq \mathrm{dom}(H)$, and $\mathsf{locs}(e') \subseteq \mathrm{dom}(H)$, we have $\mathsf{locs}(E[e']) \subseteq \mathrm{dom}(H)$.
Since $\vdash e', H$, we have $\vdash H$. Thus, by CONFIG, $\vdash E[e'], H$. □

LEMMA B.47. (Subject reduction) *If* $\vdash e, H$, $H^\dagger \vdash e : T$, *and* $e, H \longrightarrow r, H'$, *then either*

- $r = e'$, $\vdash e', H'$, *and* $H'^\dagger \vdash e' : T$, *or*
- $r = \mathsf{NullError}$.

PROOF: The proof is by induction on the typing derivation $H^\dagger \vdash e : T$.

We first consider the case where the derivation of $H^\dagger \vdash e : T$ ends with an application of T-SUB. Then $H^\dagger \vdash e : T'$ where $H^\dagger \vdash T' \leq T$.

If $r = e'$, then by the induction hypothesis, $H'^\dagger \vdash e' : T'$. By Lemma B.22, $H'^\dagger \vdash T' \leq T$. Thus, by T-SUB we can derive $H'^\dagger \vdash e' : T$.

Thus, for the remainder of the proof we need only consider typing derivations ending in a rule other than T-SUB.

We consider e by cases depending on the reduction rule used.

First, note that since $H^\dagger$ contains no $x : T$, and since $H^\dagger \vdash e : T$, e contains no free variables.

Also, note that by Lemma B.16, $H'^\dagger$ extends $H^\dagger$.

For the cases below where $e = E[e_0]$ and R-CONG applies. To show that $\vdash e, H'$, we need only show that the typing derivation for e includes $H^\dagger \vdash e_0 : T_0$. Then, by the induction hypothesis, $\vdash e_0', H'$, and by Lemma B.46, we can derive $\vdash E[e_0'], H'$.

For the cases below where $e = NE$, R-NULL applies and $r = \mathsf{NullError}$.

- $e = v$. Vacuously true since v cannot take a step.
- $e = x$. Vacuously true since e contains no free variables.
- $e = e_0.f$.
  - $e = \ell.f_i$. Then R-GET is the only rule that can apply, $H' = H$, and $r = v_i = H(\ell)[f_i]$ where $H(\ell) = S\ \{\overline{f} = \overline{v}\}$.

    Besides T-SUB, handled above, there are two cases for the derivation of $H^\dagger \vdash \ell.f_i : T$.

    - T-FIN.

      Then $T = \ell.f_i.\texttt{class}$ and f_i is a final field

    By the definition of $H^\dagger$, since $H(\ell) = S\ \{\overline{f = v}\}$, it must that $\ell.f_i.\texttt{class} = v_i.\texttt{class} \in H^\dagger$. Thus, by S-ALIAS, $H^\dagger \vdash \ell.f_i.\texttt{class} \approx v_i.\texttt{class}$.

    Thus, by T-SUB, $H^\dagger v_i \vdash \ell.f_i.\texttt{class}$.

    Note that this is the place where we use the fact that fields are final. If f_i is not final, $\ell.f_i.\texttt{class} = v_i.\texttt{class}$ will not be in $H^\dagger$.

    Since $H'^\dagger$ extends $H^\dagger$, By Lemma B.17 we have $H^\dagger v_i \vdash \ell.f_i.\texttt{class}$.

  - T-GET.

    By F-LOC and T-FIN, $H^\dagger \vdash \ell : \ell.\texttt{class}$.

    Let $\mathsf{ftype}(H^\dagger, \ell.\texttt{class}, f_i) = T_f$.

    By T-GET, $T_f = T$ and we can derive $H^\dagger \vdash \ell.f_i : T$.

    Since $\vdash H$, and $H(\ell)[f_i] = v_i$, we have by H-LOC, $H^\dagger \vdash v_i : T_f$.

- $e = \texttt{null}.f$. Then R-NULL is the only rule that can apply.
- $e = e_0.f$ where $e_0 \neq v$. Then R-CONG is the only rule that can apply and $e_0, H \longrightarrow e'_0, H'$.

  Again, there are two cases for the derivation of $H^\dagger \vdash e_0.f_i : T$.

  - T-FIN.

    Then $e_0 = p$ and $e'_0 = p'$ and $T = p.f.\texttt{class}$.

    By T-FIN, $H^\dagger \vdash_{\mathsf{final}} p.f : T_p$. By Lemma B.41 and Lemma B.42, $H = H'$, and $H^\dagger \vdash_{\mathsf{final}} p'.f : T'_p$, and $H^\dagger \vdash p.f = p'.f$. Thus, we can derive by T-FIN, $H^\dagger \vdash p'.f : p'.f.\texttt{class}$, and by S-ALIAS, $H^\dagger \vdash p.f.\texttt{class} \approx p'.f.\texttt{class}$. and by S-SUB, $H^\dagger \vdash p'.f : p.f.\texttt{class}$.

  - T-GET.

    Then $H^\dagger \vdash e_0 : T_0$ and $\mathsf{ftype}(H^\dagger, T_0, f) = T_f = T$.

    Since $H^\dagger \vdash e_0 : T_0$, by the induction hypothesis, $H'^\dagger \vdash e'_0 : T_0$.

    By Lemma B.24, we have $\mathsf{ftype}(H'^\dagger, T_0, f) = T_f$.

    Thus, we can derive by T-GET, $H'^\dagger \vdash e'_0.f : T$.

- $e = e_0.f = e_1$.
  - $e = \texttt{null}.f = e_1$. Then R-NULL is the only rule that can apply.
  - $e = \ell.f = v$. Then R-SET is the only rule that can apply and $e' = v$ and $H'(\ell)[f] = v$.

    The judgment $H^\dagger \vdash v : T$ follows trivially from T-SET.

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

    By F-LOC and T-FIN, $H'^\dagger \vdash \ell : \ell.\texttt{class}$.

    Let $\mathsf{ftype}(H^\dagger, \ell.\texttt{class}, f) = T_f$. To show that H' is well-formed, we need to show that $H'^\dagger \vdash v : T_f$.

    By T-SET, $T = T_f$ and therefore $H^\dagger \vdash v : T$.

    Therefore by Lemma B.17, $H'^\dagger \vdash v : T$.

    Since H' is equal to H except for the value stored in $H'(\ell)[f]$, namely v, and since both $H^\dagger \vdash v : T$ and $H'^\dagger \vdash v : T$, and since $H \vdash v$, it must be that $\vdash H'$.

  - $e = \ell.f = e_1$ where $e_1 \neq v$. Then R-CONG is the only rule that can apply and $e_1, H \longrightarrow e'_1, H'$.

    By T-SET, $H^\dagger \vdash \ell : T_0$, $\mathsf{ftype}(H^\dagger, T_0, f) = T_f = T$, and $H^\dagger \vdash e_1 : T$.

    By Lemma B.17, $H'^\dagger \vdash \ell : T_0$. By Lemma B.24, $\mathsf{ftype}(H'^\dagger, T_0, f) = T_f = T$,

    By the induction hypothesis $H'^\dagger \vdash e'_1 : T$.

    Thus we can derive by T-SET, $H'^\dagger \vdash e' : T$.

  - $e = e_0.f = e_1$ where $e_0 \neq v$. Then R-CONG is the only rule that can apply and $e_0, H \longrightarrow e'_0, H'$.

    By T-SET, $H^\dagger \vdash e_0 : T_0$, $\mathsf{ftype}(H^\dagger, T_0, f) = T_f = T$, and $H^\dagger \vdash e_1 : T$.

    By the induction hypothesis $H'^\dagger \vdash e'_0 : T$.

    By Lemma B.24, $\mathsf{ftype}(H'^\dagger, T_0, f) = T_f = T$, By Lemma B.17, $H'^\dagger \vdash e_1 : T$.

    Thus we can derive by T-SET, $H'^\dagger \vdash e' : T$.

- $e = e_0.m(\overline{e})$.

  By T-CALL, all of the following hold:

  - $H^\dagger \vdash e_0 : T^0_0$
  - $\mathsf{mtype}(H^\dagger, T^0_0, m) = (\overline{x} : \overline{T^0}) \to T^0_{n+1}$

- $x_0 = \texttt{this}$
- $\forall i = 1,\ldots,n+1.\ \forall j = 1,\ldots,i.\ T_i^{j-1}\{\!\{H^\dagger;\ T_{j-1}^{j-1}/x_{j-1}\}\!\} = T_i^j$
- $\forall i = 1,\ldots,n.\ \forall j = 1,\ldots,i.\ \mathsf{prefixExact}_k(T_i^{j-1}) \Rightarrow \mathsf{prefixExact}_k(T_i^j)$
- $\forall i = 1,\ldots,n.\ \forall j = 1,\ldots,i.\ p.f \in \mathsf{paths}(T_i^{j-1}) \Rightarrow p\{e_{j-1}/x_{j-1}\}.f \in \mathsf{paths}(T_i^j)$
- $\forall i = 1,\ldots,n.\ H^\dagger \vdash e_i : T_i^i$.
- $T = T_{n+1}^{n+1}$.

We consider e by cases.

- $e = \texttt{null}.m(\overline{e})$. Then R-NULL is the only rule that can apply.
- $e = \ell.m(\overline{v})$. Then R-CALL is the only rule that can apply and $H = H'$.

  By R-CALL, $H^\dagger \vdash T_0^0 \trianglelefteq S$, and $\mathsf{mbody}(S,m) = T_{n+1}\ m(\overline{T}\ \overline{x})\ \{e_m\}$. By M-OK, $\Gamma \vdash e_m : T_{n+1}$ where $\Gamma = \texttt{this} : P, \overline{x} : \overline{T}$ for some $P \in \mathsf{inh}(S)$.

  By Lemma B.17, $(H^\dagger, \Gamma) \vdash e_m : T_{n+1}$.

  Let $e_0 = e_m$ and $T_0^e = T_{n+1}$, and let $e_1 = e_m\{\ell/\texttt{this}\}$ and $T_1^e = T_{n+1}\{\ell/\texttt{this}\}$, and for $j = 1,\ldots,n$, let $e_{j+1} = e_j\{v_j/x_j\}$ and $T_{j+1}^e = T_j^e\{v_j/x_j\}$.

  Note $e' = e_{n+1}$.

  We want to show that $H^\dagger \vdash e_{n+1} : T_{n+1}^{n+1}$. We do this in two steps. First, we show (1) by Lemma B.12, $H^\dagger \vdash e_{n+1} : T_{n+1}^e$. Then we show (2) by Lemma B.14, $H^\dagger \vdash T_{n+1}^e \leq T_{n+1}^{n+1}$. By T-SUB, $H^\dagger \vdash e_{n+1} : T_{n+1}^{n+1}$.

  To apply the two lemmas, we need to show that the types of the actual values are subtypes of the (substituted) declared formal types; that is, when the lemmas are applied to a substitution of v for x in some Γ, if $x : T_x \in \Gamma$ and $\Gamma\{v/x\} \vdash v : T_v$, we must have $\Gamma\{v/x\} \vdash T_v \leq T_x$.

  First consider $x = \texttt{this}$ and $v = \ell$ in the environment $H^\dagger, \texttt{this}:P$. Since by T-CALL, $H^\dagger \vdash \ell : T_0^0$, we need to show that $H^\dagger \vdash T_0^0 \leq P$. We do so as follows: Since $H^\dagger \vdash T_0^0 \trianglelefteq S$, we have $H^\dagger \vdash T_0^0 \leq S$ by S-BOUND. Since $\vdash S \sqsubset^* P$, by Lemma B.38, $\emptyset \vdash S \leq P$. Therefore, by S-TRANS, $H^\dagger \vdash T_0^0 \leq P$.

  Now consider $x = x_1$ and $v = v_1$ in the environment $H^\dagger, \texttt{this}:P, x_1 : T_1$. By T-CALL, $H^\dagger \vdash v_1 : T_1^1$, where $T_1^1 = T_1\{\!\{H^\dagger;\ T_0^0/\texttt{this}\}\!\}$. We need to show that $H^\dagger \vdash T_1^1 \leq T_1\{\ell/\texttt{this}\}$.

  By induction on T_1.

  - $T_1 = \circ$. Then $T_1^1 = T_1\{\ell/\texttt{this}\} = T_1$.
  - $T_1 = T_1'.C$. Follows from the induction hypothesis and S-NEST.
  - $T_1 = p.\texttt{class}$.
    - $p = v$ or $p = x \neq \texttt{this}$. Then $T_1^1 = T_1\{\ell/\texttt{this}\} = T_1$.
    - $p = \texttt{this}$. Then $T_1^1 = T_0^0$ and $T_1\{\ell/\texttt{this}\} = \ell.\texttt{class}$. Since $\mathsf{exact}(T_1)$, we have $\mathsf{exact}(T_0^0)$. Therefore by S-EXACT, $H^\dagger \vdash T_0^0 \leq \ell.\texttt{class}$.
    - $p = p_0.f$. Let $T_p = p_0.\texttt{class}\{\!\{H^\dagger;\ T_0^0/\texttt{this}\}\!\}$.

      If T_p is not a path type, then $T_1^1 = \mathsf{ftype}(H^\dagger, T_p, f)$, which is not exact. Hence, this case holds vacuously.

      Otherwise, if $T_p = p_0'.\texttt{class}$, then $T_1^1 = p_0'.f.\texttt{class}$. We need to show $H^\dagger \vdash p_0'.f.\texttt{class} \leq p_0\{\ell/\texttt{this}\}.f.\texttt{class}$.

      Since T-CALL requires field paths are preserved and since $p_0.f \in \mathsf{paths}(T_1)$, we must have $p' \in \mathsf{paths}(T_1^1)$ where $({}^\dagger H) \vdash p' = p_0\{\ell/\texttt{this}\}.f$. By S-ALIAS, $H^\dagger \vdash p_0'.f.\texttt{class} \leq p_0\{\ell/\texttt{this}\}.f.\texttt{class}$.
  - $T_1 = P\texttt{[}T_1'\texttt{]}$. Follows from the induction hypothesis and S-PRE-S1.
  - $T_1 = \texttt{\&}\overline{T}$. Follows from the induction hypothesis and S-MEET-G.

  By a similar argument, we have $H^\dagger \vdash T_i^j : T_i\{\ell/\texttt{this}, v_1/x_1, \ldots, v_j/x_j\}$.

  Therefore we can apply Lemma B.12 and Lemma B.14 to show $H^\dagger \vdash e' : T_{n+1}$.

  Thus, by T-SUB, $H^\dagger \vdash e_m\{\ell/\texttt{this}, \overline{v}/\overline{x}\} : T$.
- $e = \ell.m(\overline{e})$ where some $e_i \neq v$. Then R-CONG is the only rule that can apply. WLOG let e_i be the first e_i that is not a value. Then, $e_i, H \longrightarrow e_i', H'$.

  By the induction hypothesis, $H'^\dagger \vdash e_i' : T_i^i$.

  By applying Lemma B.17 to all other subexpressions, we have for all $j \neq i$, $H'^\dagger \vdash e_j : T_j^j$ and $H'^\dagger \vdash \ell : T_0^0$.

  By Lemma B.25, $\mathsf{mtype}(H'^\dagger, T_0^0, m) = (\overline{x} : \overline{T^0}) \rightarrow T_{n+1}^0$.

By Lemma B.26, for all $j = 1, \ldots, n+1$ and all $k \leq j$, $T_j^{k-1}\{\!\{H'^\dagger;\ x_k/T_k^k\}\!\} = T_j^k$.

Since the types of all $\overline{e}$ are preserved, $\mathsf{prefixExact}_k(T_i^{j-1})$ implied $\mathsf{prefixExact}_k(T_i^j)$ before the step, then this property also holds after the step.

Since the types of all $\overline{e}$ are preserved, $\mathsf{paths}(T_i^{j-1}$ and $\mathsf{paths}(T_i^j)$ are also preserved.

Thus, we can derive by T-CALL $H'^\dagger \vdash e' : T$.

- $e = e_0.m(\overline{e})$ where $e_0 \neq v$. Then R-CONG is the only rule that can apply and $e_0, H \longrightarrow e_0', H'$.

  By the induction hypothesis, $H'^\dagger \vdash e_0' : T_0^0$.

  By Lemma B.17, we have for all $i \geq 0$ $H'^\dagger \vdash e_i : T_i^i$.

  By Lemma B.25, $\mathsf{mtype}(H'^\dagger, T_0^0, m) = (\overline{x} : \overline{T^0}) \rightarrow T_{n+1}^0$.

  By Lemma B.26, for all $j = 1, \ldots, n+1$ and all $k \leq j$, $T_j^{k-1}\{\!\{H'^\dagger;\ x_k/T_k^k\}\!\} = T_j^k$.

  Since the types of all $\overline{e}$ are preserved, $\mathsf{prefixExact}_k(T_i^{j-1})$ implied $\mathsf{prefixExact}_k(T_i^j)$ before the step, then this property also holds after the step.

  Since the types of all $\overline{e}$ are preserved, $\mathsf{paths}(T_i^{j-1}$ and $\mathsf{paths}(T_i^j)$ are also preserved.

  Thus, we can derive by T-CALL $H'^\dagger \vdash e' : T$.

- $e = \mathtt{new}\ T(\overline{f} = \overline{e})$.
  - $e = \mathtt{new}\ U(\overline{f} = \overline{v})$. Then R-NEW and R-ALLOC are the only rules that can apply.

    Let $H^\dagger \vdash U \trianglelefteq S$.
    - If $\#(\mathsf{fields}(S)) < \#(\overline{f})$, then R-NEW is the only rule that can apply and $e' = \mathtt{new}\ U(\overline{f} = \overline{v}, \overline{f'} = \overline{e'})$ and $H = H'$ and $T = U$.

      By the definition of fields, for all $f_i' \in \overline{f'}$, we have $\mathsf{ftype}(H^\dagger, U, f_i') = [\mathtt{final}]\ T_i'$

      By F-OK, for all $f_i' \in \overline{f'}$, we have $\emptyset \vdash e_i' : T_i'$. By Lemma B.17, for all i, $H'^\dagger \vdash e_i' : T_i'$.

      Thus, we can derive by T-NEW $H^\dagger \vdash e' : T$.
    - If $\#(\mathsf{fields}(S)) = \#(\overline{f})$, then R-ALLOC is the only rule that can apply and $e' = \ell$ and $H' = H, \ell \mapsto S\ \{\overline{f} = \overline{v}\}$.

      Since $H'(\ell) = S\ \{\overline{f} = \overline{v}\}$, $\ell : S \in H'^\dagger$.

      Therefore, by F-LOC, $H'^\dagger \vdash_{\mathsf{final}} \ell : S$, and by T-FIN, $H'^\dagger \vdash \ell : \ell.\mathtt{class}$.

      Since $H'^\dagger \vdash \ell.\mathtt{class} \trianglelefteq S$, we have by S-EVAL, $H'^\dagger \vdash \ell.\mathtt{class} \leq U$.

      Therefore, by S-SUB, $H'^\dagger \vdash e' : U$.

      Since $\vdash e, H$, we have $\vdash H$. Thus, $H \vdash \ell'$ for all $\ell' \in \mathrm{dom}(H)$. Since the only new location is ℓ, we just need to show that $H' \vdash \ell$ to show that Hence, $H' \vdash \ell'$ for all $\ell' \in \mathrm{dom}(H')$.

      By R-ALLOC, we have $H'(\ell) = S\ \{\overline{f} = \overline{v}\}$.

      Since $\vdash e, H$, all $\mathsf{locs}(e) \subseteq \mathrm{dom}(H)$. Therefore $\overline{v} \subseteq \mathrm{dom}(H) \cup \{\mathtt{null}\}$.

      By T-NEW, for all i, $\mathsf{ftype}(H^\dagger, U, f_i) = T_i$ and $H^\dagger \vdash v_i : T_i$.

      By Lemma B.17, for all i, $H'^\dagger \vdash v_i : T_i$.

      Thus, we can derive $H' \vdash \ell$ by H-LOC.

      Since $\ell \in \mathrm{dom}(H')$, and $e' = \ell$, we have $\mathsf{locs}(e') \subseteq \mathrm{dom}(H')$. Therefore, we can derive by CONFIG, $\vdash e', H'$.
  - $e = \mathtt{new}\ U(\overline{f} = \overline{e})$ where some $e_i \neq v$. Then R-CONG is the only rule that can apply. WLOG let e_i be the first e_i that is not a value. Then, $e_i, H \longrightarrow e_i', H'$.

    By T-NEW, $\mathsf{ftype}(H^\dagger, U, \overline{f}) = \overline{T}$. By Lemma B.24, we have $\mathsf{ftype}(H'^\dagger, U, \overline{f}) = \overline{T}$.

    By T-NEW, $H^\dagger \vdash e_i : T_i$. Therefore, by the induction hypothesis, $H'^\dagger \vdash e_i' : T_i$.

    With this judgment and by Lemma B.17 for all other subexpressions, we have $H'^\dagger \vdash \overline{e} : \overline{T}$.

    Thus, by T-NEW, we can derive $H'^\dagger \vdash e' : T$.
  - $e = \mathtt{new}\ TE[\mathtt{null}](\overline{f} = \overline{e})$. Then R-NULL is the only rule that can apply.
  - $e = \mathtt{new}\ TE[p](\overline{f} = \overline{e})$ where $p \neq \mathtt{null}$ and $TE[p] \neq U$. Then R-CONG is the only rule that can apply and $p, H \longrightarrow p', H$.

    By T-NEW, we have $H^\dagger \vdash \overline{e} : \overline{T}$.

    By Lemma B.17, we have $H'^\dagger \vdash \overline{e} : \overline{T}$.

    Since $H^\dagger \vdash TE[p]$, by Lemma B.45, $H^\dagger \vdash TE[p']$.

    Thus, by T-NEW, we can derive $H'^\dagger \vdash e' : TE[p']$.
- $e = e_1;\ e_2$.

  - $e = v_1;\ e_2$. Then R-SEQ is the only rule that can apply, and $H = H'$ and $r = e_2$.
    By T-SEQ, since $H^\dagger \vdash v_1;\ e_2 : T$, we have $H^\dagger \vdash e_2 : T$.
    Since $H = H'$, $H^\dagger \vdash e_2, H$.
  - $e = e_1;\ e_2$ where $e_1 \neq v$. Then R-CONG is the only rule that can apply and $r = e'_1;\ e_2$.
    By T-SEQ, since $H^\dagger \vdash e_1;\ e_2 : T$, we have $H^\dagger \vdash e_1 : T_1$ and $H^\dagger \vdash e_2 : T$.
    By the induction hypothesis, $H'^\dagger \vdash e'_1 : T_1$. By Lemma B.17, $H'^\dagger \vdash e_2 : T$. Thus we can derive, by T-SEQ, $H'^\dagger \vdash e'_1;\ e_2 : T$.

□

B.7 Progress

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

LEMMA B.48. (Progress) *If* $\vdash e,H$ *and* $H^\dagger \vdash e : T$, *then either* $e = v$, *or there is an* r *and an* H' *such that* $e,H \longrightarrow r,H'$.

PROOF: By structural induction on e.

- $e = \texttt{null}$. Trivial since e is a value.
- $e = \ell$. Trivial since e is a value.
- $e = x$. Vacuous since $H^\dagger \nvdash x : T$.
- $e = e_0.f$.
  - If $e_0 = \texttt{null}$, then the configuration can take a step by R-NULL.
  - If $e_0 = \ell$, then since $\vdash e,H$, $H(\ell) = S\ \{\overline{f} = \overline{v}\}$ and $f \in \overline{f}$, and so the configuration can take a step by R-GET.
  - Otherwise, e can take a step by R-CONG.
- $e = e_0.f = e_1$.
  - If $e_0 = \texttt{null}$, then the configuration can take a step by R-NULL.
  - If $e_0 = \ell$ and $e_1 = v$, then since $\vdash e,H$, $H(\ell) = S\ \{\overline{f} = \overline{v}\}$ and $f \in \overline{f}$, and so the configuration can take a step by R-SET.
  - Otherwise, e can take a step by R-CONG.
- $e = e_0.m(\overline{e})$.
  - If $e_0 = \texttt{null}$, then the configuration can take a step by R-NULL.
  - Assume $e_0 = \ell$ and $\overline{e}$ are all values. Since $\vdash e,H$, $\ell{:}S \in H^\dagger$ for some S. Therefore, $H^\dagger \vdash \ell{:}S$ by F-LOC and T-FIN. Since $H^\dagger \vdash e{:}T$, by T-CALL $\mathsf{mtype}(H^\dagger, S, m)$ is defined. Since $\emptyset \vdash S$, $\mathsf{mtype}(\emptyset, S, m) = \mathsf{mtype}(H^\dagger, S, m)$. Hence, by Lemma B.40, $\mathsf{mbody}(S, m)$ is defined and, therefore, a step can be taken by R-CALL.
  - Otherwise, e can take a step by R-CONG.
- $e = \texttt{new}\ T(\overline{f} = \overline{e})$.
  - If $T = U$, and $\overline{e}$ are all values, then since $\vdash e,H$, there is an S such that $H^\dagger \vdash U \trianglelefteq S$. If $\#(\mathsf{fields}(S)) = \#(\overline{f})$, a step can be taken by R-ALLOC; otherwise, if $\#(\mathsf{fields}(S)) < \#(\overline{f})$, a step can be taken by R-NEW.
  - Otherwise, e can take a step by R-CONG.
- $e = e_1;\ e_2$. If $e_1 = v$, a step can be taken by R-SEQ. Otherwise, e can take a step by R-CONG.

□

B.8 Soundness

Soundness follows directly from the subject reduction and progress lemmas.

THEOREM B.49. (Soundness) *If* $\vdash \langle \overline{L}, e \rangle$ ok *and* $\emptyset \vdash e{:}T$, *then there is an* r *such that* $e, \emptyset \longrightarrow^* r, H'$. *and* $r = v$ *and* $H'^\dagger \vdash v{:}T$ *or* $r = \mathsf{NullError}$,

PROOF: Follows from Lemma B.47 and Lemma B.48. □