Morphing: Structurally Shaping a Class by Reflecting on Others

Shan Shan Huang
LogicBlox, Inc
Yannis Smaragdakis
University of Massachusetts, Amherst

We present MorphJ: a language for specifying general classes whose members are produced by iterating over members of other classes. We call this technique "class morphing" or just "morphing". Morphing extends the notion of genericity so that not only types of methods and fields, but also the *structure* of a class can vary according to type variables. This adds a disciplined form of meta-programming to mainstream languages and allows expressing common programming patterns in a highly generic way that is otherwise not supported by conventional techniques. For instance, morphing lets us write generic proxies (i.e., classes that can be parameterized with another class and export the same public methods as that class); default implementations (e.g., a generic do-nothing type, configurable for any interface); semantic extensions (e.g., specialized behavior for methods that declare a certain annotation); and more. MorphJ's hallmark feature is that, despite its emphasis on generality, it allows modular type checking: a MorphJ class can be checked independently of its uses. Thus, the possibility of supplying a type parameter that will lead to invalid code is detected early—an invaluable feature for highly general components that will be statically instantiated by other programmers. We demonstrate the benefits of morphing with several examples, including a MorphJ reimplementation of DSTM2, a software transactional memory library, which reduces 1,484 lines of Java reflection and bytecode engineering library calls to just 586 lines of MorphJ code.

Categories and Subject Descriptors: D.1.2 [**Programming Techniques**]: Automatic Programming—*program synthesis, program transformation*; D.3.3 [**Programming Languages**]: Language Constructs and Features; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Design, Languages

Additional Key Words and Phrases: meta-programming, language extensions, morphing

1. INTRODUCTION

Consider the following task: how would you write a piece of code that, given *any* class `X`, returns another class that contains the exact same methods as `X`, but logs each method's return value? That is, the code is a reusable representation of the

Author's address: S. Huang, LogicBlox, Inc., Two Midtown Plaza, Suite 1880 1349 West Peachtree Street, N.W. Atlanta, GA 30309
Y. Smaragdakis, University of Massachusetts, Amherst, Department of Computer Science, 140 Governors Drive, University of Massachusetts, Amherst, MA 01003

functionality "logging", and abstracts over the exact structure of the class (i.e., its declared methods) it may be applied to.

Capturing this level of abstraction has traditionally been only possible with techniques such as meta-object protocols (MOPs) [Kiczales et al. 1991], aspect-oriented programming (AOP) [Kiczales et al. 1997], or various forms of meta-programming (e.g., reflection and ad-hoc program generation using quote primitives, string templates, or bytecode engineering). While just about any programmer can write a method that logs its return value, the techniques listed above can require steep learning curves or suffer from poor integration with the base language. More importantly, these techniques do not offer any modular safety guarantee: There is no guarantee that a piece of code would always be well-typed regardless of its uses in specific compositions.

In this article we discuss a general approach called *morphing* for writing such structurally abstract code, while maintaining modular safety guarantees. We demonstrate the power of morphing through MorphJ—a reference language that demonstrates what we consider the desired expressiveness and safety features of an advanced morphing language. MorphJ can express highly reusable object-oriented components (i.e., generic classes) whose exact members (e.g., fields and methods) are not known until the component is parameterized with concrete types. For instance, the following MorphJ class implements the "logging" extension described above:

```
1 class Logging<class X> extends X {
2   <R,Y*>[meth]for(public R meth (Y) : X.methods)
3   public R meth (Y a) {
4     R r = super.meth(a);
5     System.out.println("Returned: " + r);
6     return r;
7   }
8 }
```

MorphJ allows class `Logging` to be declared as a subclass of its type parameter, `X`. The body of `Logging` is defined by static iteration (using the `for` statement, on line 2) over all methods of `X` that match the pattern "`public R meth(Y)`". `Y`, `R`, and `meth` are pattern-matching variables. `Y` and `R` can match any non-`void` type, and `meth` can match any identifier. Additionally, the `*` symbol following the declaration of `Y` indicates that `Y` can match a vector of any number of types (including zero). That is, the above pattern matches all `public` methods that return any non-`void` type. The pattern-matching variables are also used in the declaration of `Logging`'s methods: for each method of the type parameter `X` matched, `Logging` declares a method with the same name and type signature. (This does not have to be the case, as shown later.) Thus, the exact methods of class `Logging` are not determined until it is composed, or type-instantiated, with a concrete type. For instance, `Logging<java.lang.Object>` has methods `equals`, `hashCode`, and `toString`: these are the only `public`, non-`void`-returning methods of `java.lang.Object`.

The above example illustrates the basic feature of MorphJ: code can be declared by iterating over members of a class or interface matching a pattern. MorphJ also allows even more expressive iteration conditions using *nested patterns*. For instance,

the following code iterates over the `public`, non-`void`-returning methods of `X`, *such that there is also some method in* `X` *with the same name, taking no argument*:

```
<R,Y*>[meth]for ( public R meth(Y) : X.methods;
                  some R meth() : X.methods ) ...
```

The *positive nested pattern*, "`some R meth() : X.methods`", places a positive existential condition on the outer, primary pattern. A negative existential condition can be similarly imposed on the primary pattern by annotating the nested pattern with the keyword `no`, instead of `some`, making it a *negative nested pattern*.

"Reflective" program pattern matching and transformation, as in the logging example, are not new. Several pattern matching languages have been proposed in prior literature (e.g., [Bachrach and Playford 2001; Baker and Hsieh 2002; Batory et al. 1998; Visser 2004]) and most of them specify transformations based on some intermediate program representation (e.g., abstract syntax trees) although the patterns resemble regular program syntax. Nevertheless, MorphJ elevates reflective transformation functionality to a disciplined language feature, and expresses it as an extension of simple genericity. An important aspect of full-fledged language support is modularity: MorphJ generic classes support separate type checking—a generic class is type-checked independently of its type-instantiations, and errors are detected if they can occur with *any* possible type parameter. For an example of separate type checking, consider a "buggy" generic class:

```
class CallWithMax<class X> extends X {
  <A>[m]for(public int m (A) : X.methods)
  int m(A a1, A a2) {
    if (a1.compareTo(a2) > 0)
      return super.m(a1);
    else
      return super.m(a2);
  }
}
```

The intent is that class `CallWithMax<C>`, for some `C`, imitates the interface of `C` for all single-argument methods that return `int`, yet adds an extra formal parameter to each method. The corresponding method of `C` is then called with the greater of the two arguments passed to the method in `CallWithMax<C>`. It is easy to define, use, and deploy such a generic transformation without realizing that it is not always well-typed: not all types `A` will support the `compareTo` method. MorphJ detects such errors when compiling the above code, independently of specific type-instantiations. In this case, the fix is to strengthen the pattern with the constraint that `A` must be a subtype of `Comparable<A>`:

```
<A extends Comparable<A>>[m]for(public int m (A) : X.methods)
```

Additionally, the above code has an even more insidious error. The generated methods in `CallWithMax<C>` are not guaranteed to correctly override the methods in its superclass, `C`. For instance, if `C` contains two methods, `int foo(int)` and `String foo(int,int)`, then the latter will be improperly overridden by the generated method `int foo(int,int)` in `CallWithMax<C>` (which has the same argument types but an incompatible return type). MorphJ statically catches this error. This is an

instance of the complexity of MorphJ's modular type checking when dealing with unknown entities.

Separate type-checking is an invaluable property for generic code: It prevents errors that only appear for some type parameters, which the author of the generic class may not have predicted. Accordingly, it allows modular reasoning when developing morphed classes: the type system ensures that the class specifies all its assumptions about the type parameters it will be used with. Effectively, morphing adds static typing to reflective class generation and transformation. This is a substantial improvement that makes morphing an important new approach to software development. The present article is intended as a comprehensive reference for morphing, its techniques, its impact, and its potential. The article collects and expands the earlier MorphJ publications [Huang et al. 2007b; Huang and Smaragdakis 2008], offering an integrated view of morphing features, complete with applications and with a formal treatment of the MorphJ type system.

In the remainder of the article, Section 2 introduces basic MorphJ language features; Section 3 gives real-world examples in which basic MorphJ features increases the modularity and reusability of software components; Section 4 introduces advanced MorphJ features such as nested patterns; Section 5 provide two real-world examples using nested patterns. Section 6 gives a gentle and casual overview of MorphJ's type system, while Section 7 formalizes a core subset of MorphJ and presents the type rules and a soundness proof. We briefly discuss the implementation of MorphJ in Section 8.2 and give a survey of related work in Section 9. We conclude with our grand-scheme view of the evolution of programming languages and the role of MorphJ and other related mechanisms (Section 10).

2. BASIC MORPHJ FEATURES

MorphJ adds to Java the ability to include *reflective iteration blocks* inside a class or interface declaration. The purpose of a reflective iteration block is to *statically iterate* over a certain subset of a type's methods or fields, and produce a declaration or statement for each element in the iterator. Static iteration means that no runtime reflection exists in compiled MorphJ programs. All declarations or statements within a reflective block are "generated" at compile-time.

A reflective iteration block (or reflective block) has similar syntax to the existing `for` iterator construct in Java. There are two main components to a reflective block: the iterator definition, and the code block for each iteration. The following is a MorphJ class declaration with a very simple reflective block:

```
class C<T> {
  for ( static int foo () : T.methods ) {|
    public String foo () { return String.valueOf(T.foo()); }
  |}
}
```

We overload the keyword `for` for static iteration. The iterator definition immediately follows `for`, delimited by parentheses. The iterator definition consists of just a *pattern* (we will later generalize this to allow more patterns) with the format "*signature pattern* : *reflection set*". The *reflection set* is defined by applying the `.methods` or `.fields` keywords to a type, designating all methods or fields of that

type. The *signature pattern* is either a method or field signature, possibly with type and name variables, used to filter out elements from the reflection set. We call the set of elements of the reflection set that match the signature pattern the *reflective range* (or just *range*) of the pattern or the iterator. In the example above, the reflective range contains only `static` methods of type `T`, with name `foo`, no argument, and return type `int`.

The second component of a reflective block is delimited by {|...|}, and contains either method or field declarations or a block of statements. The reflective block is itself syntactically a declaration or block of statements, and can be used either at the class members level (to define methods and fields) or inside the body of a method (to define statements). We prevent reflective blocks from nesting. In case of a single declaration (as in most examples in this article), the delimiters can be dropped. The declarations or statements are "generated", once for each element in the reflective range of the block. In the example above, a method `public String foo() { ... }` is declared for each element in the reflective range. Thus, if `T` has a method `foo` matching the signature pattern `static int foo()`, a method `public String foo()` exists for class `C<T>`, as well.

The reflective block in the previous example is rather boring. Its reflective range contains at most one method, and we know statically the type and name of that method. For more flexible patterns, we can introduce type and name variables for pattern matching. Pattern matching type and name variables are defined right before the `for` keyword. They are only visible within that reflective block, and can be used as regular types and names. For example:

```
class C<T> {
  T t;
  C(T t) { this.t = t; }

  <A>[m] for (int m (A) : T.methods )
  int m (A a) { return t.m(a); }
}
```

The above signature pattern matches methods of *any* name that take one argument of *any* type and return `int`. The matching of multiple names and types is done by introducing a type variable, `A`, and a name variable, `m`. Name variables match *any* identifier and are introduced by enclosing them in `[...]`. The syntax for introducing pattern matching type variables extends that for declaring type parameters for generic Java classes: new type variables are enclosed in `<...>`. We can give type variable `A` one or more bounds (e.g., `<A extends Foo & Bar>`), and the bounds can contain `A` itself (e.g., `<A extends Comparable<A>>`). Multiple type variables can be introduced, as well: `<A extends Foo,B extends Bar>`. In addition to the Java generics syntax, we can annotate a type parameter with keywords `class` or `interface`. For instance `<interface A>` declares a type parameter `A` that can only match an interface type. (This extension also applies to non-pattern-matching type parameters, in which case `A` can only be instantiated with an interface.) A semantic difference between pattern matching type parameters and type parameters in Java generics is that a pattern matching type parameter is not required to be a non-primitive type. In fact, without any declared bounds or `class`/`interface` keyword,

`A` can match any type that is not `void`—this includes primitive types such as `int`, `boolean`, etc. To declare a type variable that only matches non-primitive types, one can write `<A extends Object>`. Many of the above distinctions stem from low-level (sometimes just concrete syntax) differences in Java. E.g., a `void`-returning method does not need a `return` statement, or a class cannot extend an interface.

The type and name variables declared for the reflective block can be used as regular types and names inside the block. In the example above, a method is declared for each method in the reflective range, and each declaration has the same name and argument types as the method that is the current element in the iteration. The body of the method calls method `m` on a variable of type `T`—whatever the value of `m` is for that iteration, this is the method being invoked.

Note that a pattern *cannot* iterate over methods or fields of a pattern-matching type variable. One can only iterate over members of concrete types, or class/interface level type variables.

Often, a user does not care (or know) how many arguments a method takes. It is only important to be able to faithfully replicate argument types inside the reflective block. We provide a special syntax for matching *any* number of types: a `*` suffix on the pattern matching type variable definition. For instance, if a pattern matching type variable is declared as `<A*>`, then `String m (A)` is a signature pattern that matches any method returning `String`, no matter how many arguments it takes (including zero arguments), and no matter what the argument types are. Even though `A*` is technically a vector of types, it can only be used as a single entity inside of the reflective block. MorphJ provides no facility for iterating over the vector of types matching `A`. This relieves us from having to deal with issues of order or length.

MorphJ also offers the ability to construct *new* names from a name variable, by prefixing the variable with a constant. MorphJ provides the construct `#` for this purpose. To prefix a name variable `f` with the static name `get`, the user writes `get#f`. Note that the prefix cannot be another name variable—e.g., `get` is a literal prefix. Creating names out of name variables can cause possible naming conflicts. In later sections, we discuss in detail how the MorphJ type system ensures that the resulting identifiers are unique. MorphJ also offers the ability to create a string out of a name variable (i.e., to use the name of the method or field that the variable currently matches as a string) via the syntax *`var`*`.name`. The example below demonstrates these features:

```
class C<T> {
  T t;
  C(T t) { this.t = t; }

  <R,A*>[m] for (public R m (A) : T.methods )
  R delegate#m (A a) {
    System.out.println("Calling method "+ m.name + " on " + t.toString());
    return t.m(a);
  }
}
```

The above example shows a simple proxy class that declares methods that mimic

the (non-`void`-returning) `public` methods of its type parameter. Declared method names are the original method names prefixed by the constant name `delegate`. Declared methods call the corresponding original methods after logging the call.

In addition to the above features, MorphJ also allows matching arbitrary modifiers (e.g., `final`, `synchronized` or `transient`), exception clauses, and Java annotations. (Annotation matching is particularly important in practice, since Java annotations are the mainstream way for smooth syntactic extension of the language.) MorphJ has a set of conventions to handle modifier, exception, and annotation matching so that patterns are not burdened with unnecessary detail—e.g., for most modifiers, a pattern that does not explicitly mention them matches regardless of their presence, and absence of a modifier is designated by prefixing it with `!`. We do not elaborate further on these aspects of the language, as they represent merely engineering conveniences and are orthogonal to the main MorphJ insights: the morphing language features, combined with a modular type-checking approach.

3. APPLICATIONS USING BASIC MORPHING FEATURES

Even with the basic features introduced in the previous section, MorphJ opens the door for expressing a large number of useful idioms in a general, reusable way. We next show three applications using MorphJ that demonstrate the power of its structural abstraction mechanism.

3.1 Generic Synchronization Proxy

The Java Collections Framework (JCF) is the standard data structure library of the Java language. JCF defines a number of synchronization proxy classes for its main data structure interfaces. For instance, Figure 1(a) shows the definition of `SynchronizedCollection<E>`, a synchronization proxy for the interface `Collection<E>`. For each method of `Collection<E>`, `SynchronizedCollection<E>` defines a method with the same signature. Within each method body, a mutex is first acquired, and the call is delegated to the underlying `Collection<E>` object.

The definition of `SynchronizedCollection<E>` exhibits two levels of structural redundancy. At the method level, every method shares the exact same structure, with variations only in the method a call is delegated to, and the arguments used in the delegation. Another level of redundancy exists at the class level. The code for `SynchronizedCollection<E>` represents a fixed composition of the "synchronization" behavior with the data structure `Collection<E>`. To synchronize a different data type, a new class needs to be defined. For instance, Figure 1(b) is the definition of `SynchronizedList<E>`, the fixed composition of "synchronization" with `List<E>`. The definition of `SynchronizedList<E>` shares the exact same structural pattern as `SynchronizedCollection<E>`: A method is declared for each method of `List<E>`, and each method acquires a mutex before delegating the call.

JCF defines four more such synchronization proxies: `SynchronizedSet`, `SynchronizedSortedSet`, `SynchronizedRandomAccessList`, `SynchronizedMap`, and `SynchronizedSortedMap`. Using MorphJ, we can implement a highly generic class `Synchronized<X>` (Figure 2) to replace all `Synchronized*` classes in JCF. The MorphJ class can also be reused with many more data structures that may desire the synchronization behavior.

```
class SynchronizedCollection<E> implements Collection<E> {
  Collection<E> c;
  Object mutex;
  ... // constructors that initialize c and mutex

  public int size() {
    synchronized(mutex) { return c.size(); }
  }
  public boolean remove(E e) {
    synchronized(mutex) { return c.remove(e); }
  }
  ... // repeat for all methods in Collection.
}
```

(a) Synchronization proxy for `Collection`.

```
class SynchronizedList<E> implements List<E> {
  List<E> l;
  Object mutex;
  ... // constructors that initialize l and mutex

  public int size() {
    synchronized(mutex) { return l.size(); }
  }
  public int indexOf (E e) {
    synchronized(mutex) { return l.indexOf(o); }
  }
  ... // repeat for all methods in List.
}
```

(b) Synchronization proxy for `List`.

Fig. 1. Definition of synchronization proxies in JCF.

The MorphJ class `Synchronized` is a subtype of its type parameter, `X`. Additionally, `X` is required to be an interface, not a class. The body of `Synchronized` contains two reflective iteration blocks: lines 6-9, and lines 11-14. The block on lines 6-9 iterates over all non-`void` returning method of the type parameter `X` . For each matching method in `X`, a method with the same name and type signature is declared in `Synchronized<X>`. The body of the method synchronizes on a mutex first, and then delegates the method call to the underlying `X` object. The reflective iteration block on lines 11-14 declares similar methods for the `void` returning methods of `X`. Thus, `Synchronized<Collection<E>>` is the functional equivalent of the hardcoded proxy `SynchronizedCollection<E>` we saw in Figure 1.

The full version of the above MorphJ class[1] consists of less than 26 lines of code, replacing more than 314 lines of `Sychronized*` class definitions in the JCF.

[1]The full version of the MorphJ class declares constructors that allows the caller of a constructor

```
1  public class Synchronized<interface X> implements X {
2    X      me;
3    Object mutex;
4    // ... constructor declarations
5
6    <R,A*>[m] for (public R m (A) : X.methods)
7    R m (A args) {
8      synchronized (mutex) { return me.m(args); }
9    }
10
11   <A*>[m] for (public void m (A) : X.methods)
12   void m (A args) {
13     synchronized (mutex) { me.m(args); }
14   }
15 }
```

Fig. 2. A highly generic `Synchronized<X>` class in MorphJ

Compared to classes such as `SynchronizedCollection<E>`, `Synchronized<X>` is more general: It can be composed with any interface needing the "synchronization" behavior. (A similar generic class can be defined to add the behavior to classes.) Furthermore, `Synchronized<X>` is also immune to interface changes in the interface of `X`. When a method is added to or deleted from `X`, or when a method signature changes, the structure of `Synchronized<X>` automatically adapts.

3.2 Default Class

Consider a general "default implementation" class that adapts its contents to any interface used as a type parameter. The class implements all methods in the interface, with each method implementation returning a default value. Figure 3 shows such a MorphJ generic class. (Note that the keyword `throws` in the pattern does not prevent methods with no exceptions from being matched, since `E` is declared to match a possibly-zero length vector of types.)

This functionality is particularly useful for testing purposes—e.g., in the context of an application framework (where parts of the hierarchy will be implemented only by the end user), in uses of the Strategy pattern [Gamma et al. 1995] with "neutral" strategies, etc. For instance, the Java Swing framework (a standard Java GUI library) defines classes with default no-action methods for all event listener interfaces (such as `MouseListener`, `MouseMotionListener`). All such default no-action classes can be replaced by employing the morphed class of Figure 3.

One can easily think of ways to enrich this example with more complex default behavior, e.g., returning random values or calling constructor methods, instead of using statically determined default values. The essence of the technique, however, is in the iteration over existing methods and special handling of each case of return type. This is only possible because of morphing capabilities. In practice, random testing systems (e.g., [Csallner and Smaragdakis 2004]) often implement

to set the mutex to either `this`, or an arbitrary object.

```
class DefaultClass<interface I> implements I {
  // For each method returning a non-primitive type,
  // make it return null
  <R extends Object,A*,E*>[m] for( R m (A) throws E : T.methods )
  public R m ( A a ) throws E { return null; }

  // For each method returning a primitive type,
  // return a default value
  <A*,E*>[m]for( int m (A) throws E : T.methods )
  public int m (A a ) throws E { return 0; }

  ... // repeat the above for each primitive return type.

  // For each method returning void, simply do nothing.
  <A*,E*>[m] for ( void m (A) throws E : T.methods )
  public void m (A a) throws E { }
}
```

Fig. 3. A generic MorphJ class providing default implementations for every method of its type parameter interface.

very similar functionality using unsafe run-time reflection. Errors in the reflective or code generating logic are thus not caught until they are triggered by the right combination of inputs, unlike in the MorphJ case.

3.3 Sort-by

A common scenario in data structure libraries is that of supporting sorting according to different fields of a type. Although one can use a generic sorting routine that accepts a comparison function, the comparison function needs to be custom-written for each field of a type of interest. Instead, a simpler solution is to morph comparison functions based on the fields of a type.

Consider the implementation of an `ArrayList` in Figure 4, modeled after the `ArrayList` class in the Java Collections Framework. `ArrayList<E>` supports a method `sortBy#f` for every field `f` of type `E`. The power of the above code does not have to do with comparing elements of a certain *type* (this can be done with existing Java generics facilities), but with calling the comparison code on the exact fields that need it. For instance, a crucial part that is not expressible with conventional techniques is the code `e1.f.compareTo(e2.f)` (line 11), for any field `f`.

The above examples illustrate the power of MorphJ's morphing features: a generic class or interface can be shaped by the properties of the members of the type it is composed with. The morphing approach is similar to reflection, yet all reasoning is performed statically, there is syntax support for easily creating new fields and methods, and type safety is statically guaranteed.

Even more examples from the static reflection or generic aspects literature [Draheim et al. 2005; Fähndrich et al. 2006; Huang et al. 2005; Kiczales et al. 2001] can be viewed as instances of morphing and can be expressed in MorphJ. For instance, the CTR work [Fähndrich et al. 2006] allows the user to express a "transform"

```
1  public class ArrayList<E> extends AbstractList<E>
2    implements List<E>,RandomAccess,Cloneable,java.io.Serializable {
3    ...// ArrayList fields and methods.
4
5    // For each Comparable field of E, declare a sortBy method
6    <F extends Comparable<F>>[f]for(public F f : E.fields)
7    public void sortBy#f () {
8      Collections.sort(this,
9                       new Comparator<E> () {
10                        public int compare(E e1, E e2) {
11                          return e1.f.compareTo(e2.f);
12                        }
13                      });
14   }
15 }
```

Fig. 4. An ArrayList implementation providing sorting methods by each comparable field of its element type.

that iterates over methods of a class that have a `@UnitTestEntry` annotation and generate code to call all such methods while logging the unit test results. The same example can be expressed in MorphJ, with some advantages over CTR: MorphJ is better integrated in the language, using generic classes instead of a "transform" concept; MorphJ is a more expressive language, e.g., allowing matching methods with an arbitrary number and types of arguments; MorphJ offers stronger guarantees of modular type safety, as its type system detects the possibility of conflicting definitions (CTR only concentrates on preventing references to undefined entities), and we offer a proof of type soundness (Section 7).

4. ADVANCED MORPHING WITH NESTED PATTERNS

Though we have shown a number of applications of reflective declaration of methods using simple patterns, there are many morphing tasks that require the mechanism of *nested patterns* in order to be expressed type-safely. A nested pattern has the same syntactic form as the single (*primary*) patterns we have seen so far, but is preceded by the keywords "`some`" (for a *positive* nested pattern) or "`no`" (for a *negative* nested pattern). Like primary patterns, nested patterns can only reflect over concrete types, or type variables of the generic class. A nested pattern places a condition (nested condition) on each element matched by the primary pattern. An element must be matched by the primary pattern *and* satisfy all nested conditions to be a part of a reflective block's range. We next illustrate the need for and uses of nested patterns.

4.1 Negative Nested Pattern

A negative nested pattern exerts a condition that is only satisfied if there is *nothing* in the range of the pattern.

To see the necessity for negative nested patterns, consider the pesky problem of defining "getter" methods for fields in a class. Currently, programmers deal with

this by repeating the same boiler-plate code for each field. This seems to be a task perfectly suited for pattern-based reflective declaration. However, we cannot implement this in a type-safe way using the basic pattern-based feature of MorphJ. Consider the following attempt:

```
class AddGetter<class X> extends X {
  <F>[f] for ( F f : X.fields )
  F get#f () { return super.f; }
}
```

`AddGetter<X>` defines a method `get#f()` for each field `f` in type variable `X`. `get#f` denotes an identifier that begins with the string "`get`", followed by the identifier matched by `f`.[2] However, `AddGetter<X>` is not modularly type safe—we cannot guarantee that, no matter what `X` is instantiated with, `AddGetter<X>` is always well-typed. Suppose we have class `C`:

```
class C {
  Meal lunch;  ... // other methods.
  boolean getlunch() { return isNoon() ? true : false; }
}
```

With a type-instantiation `AddGetter<C>`, a client would rely on the existence of method `getlunch` and would be able to call it, expecting to receive a `Meal` object. Nevertheless, method "`Meal getlunch()`" in `AddGetter<C>` would incorrectly override method "`boolean getlunch()`" in its superclass, `C`. For this reason, the definition of `AddGetter<X>` does not pass MorphJ's type-checking.

This is an error of under-specified requirements. The definition of `AddGetter<X>` should clearly specify that it can only declare method `get#f` for those fields `f` for which a conflictingly defined `get#f` *does not already exist* in `X`. What we need is to place a *negative existential condition* on each field matched by the pattern: for all fields `f` of `X` such that method `get#f()` does not already exist in `X`, declare the method `get#f()`.

Negative nested patterns give us precisely the ability to specify such negative existential conditions. The following is a modularly type safe implementation of `AddGetter<X>`:

```
1 class AddGetter<X> extends X {
2   <F>[f]for( F f : X.fields ; no get#f() : X.methods )
3   F get#f() { return super.f; }
4 }
```

The nested pattern condition on line 2 is only satisfied by those fields `f` of `X` for which there is no method `get#f()` in `X`. (The missing return type in the nested pattern is a MorphJ shorthand for matching both `void` and non-`void` return types.) Observe that this `AddGetter<X>` class will not introduce ill-typed code for *any* `X`.

[2]This MorphJ class only defines getter methods for non-`private` fields: the semantics of pattern "`F f`" without modifier specification is that it matches all non-`private` fields. This is a reasonable default since a morphed class can at most be in a subtype relationship with its type parameter—i.e., MorphJ currently only supports class extension by inheritance. Such example code applies to cases when a getter method is required in order to follow external conventions (e.g., conventions for Enterprise Java Beans).

Potentially conflicting method declarations are prevented by the negative nested pattern. A field `f` for which a method `get#f()` already exists in `X` does not satisfy the nested condition, and thus is not in the range of the reflective block.

4.2 Positive Nested Pattern

A positive nested pattern exerts a condition that is only satisfied if there is *some* (i.e., at least one) element in the range of the pattern.

To see when positive nested patterns may be useful, consider how one would define a class `Pair<X,Y>`, which is a container for objects `x` and `y` of types `X` and `Y`, respectively. For every non-`void` method that `X` and `Y` have in common (i.e., same method name and argument types), `Pair<X,Y>` should declare a method with the same name and argument types, but a return type that is another `Pair`, constructed from the return types of that method in `X` and `Y`.

In order to express this type of functionality, we need a *positive existential condition*: for all methods in `X`, such that another method with the same name and argument types exists in `Y`, declare a method that invokes both and returns a `Pair` of their values.

With a positive nested pattern, we can define the `Pair<X,Y>` class as shown in Figure 5. Methods of `Pair<X,Y>` are defined using the reflective block on lines 5-10. The primary pattern on line 6 matches all non-`void` and non-primitive methods of `X`. For each such method, the positive nested condition on line 7 is only satisfied if a method with the same name and argument types also exists in `Y`. Thus, the primary and nested patterns in this class find precisely all methods that `X` and `Y` share in name and argument types. For each such method, `Pair<X,Y>` declares a method with the same name and argument types, and a body that invokes the corresponding method of `X` and `Y`. The return type is another `Pair`, constructed from the return values of the invocations. Following the same pattern, the class can be enhanced to also handle methods returning primitive types or `void`.

```
1  public class Pair<X,Y> {
2     X x; Y y;
3     public Pair(X x, Y y) { this.x = x; this.y = y; }
4
5    <RX extends Object, RY extends Object, A*>[m]
6    for(   public RX m(A) : X.methods ;
7      some public RY m(A) : Y.methods )
8    public Pair<RX,RY> m(A args) {
9       return new Pair<RX,RY>(x.m(args), y.m(args));
10   }
11 }
```

Fig. 5. A Pair container class using positive nested patterns.

4.3 More Features: `if`, `errorif`

Nested patterns enable other powerful language features. The reflective declarations we have seen so far have been iteration-based: a piece of code is declared for each element in the range. MorphJ also supports condition-based reflective declarations and statements. (Section 4.4 explains precisely how nested patterns enable condition-based reflective declarations.) An example (from an application discussed in detail in Section 5) illustrates the usage of pure conditions in reflective declarations:

```
<R> if ( no public R restore() : X.methods )
public void restore() { ... }
```

The above reflective declaration block consists of a statically exerted condition, specified by the pattern following the `if` keyword. If the pattern condition is satisfied, the code following the condition is declared. Thus, method `void restore()` is only declared if a method `restore()`, with any non-`void` return type, does not already exist in `X`.

Another useful feature is introduced by the `errorif` keyword. `errorif` acts as a type-assertion, allowing the programmer to express facts that he/she knows to be true about a type parameter. For instance, the programmer can express a mixin class that is only applicable to non-conflicting classes:

```
class SizeMixin<X> extends X {
  <F> errorif ( some F size : X.fields )
  int size = 0;
}
```

In this case, the programmer wants to assert that, if the parameter type `C` already contains a field named `size`, this is not an error in the definition of `SizeMixin` but in the instantiation `SizeMixin<C>`. Thus, the `errorif` construct serves as a typical type-cast: it is both an assumption that the type system can use (i.e., when checking `SizeMixin<X>` it can be assumed that `X` has no `size` field) and at the same time a type obligation for a later type-checking stage. Unlike, however, traditional type-casts that turn a static type check into a run-time type check, an `errorif` turns a modular type check into a non-modular (type-instantiation time), but still static, type check. Most of the negative nested pattern examples we present in this article (e.g., in Section 4.1 but also later in Section 5) can be restated with an `errorif`, to cause the negative condition to result in an error, instead of problematic conflicts being skipped.

4.4 Semantics of Nested Patterns

Similarly to primary patterns, a nested pattern introduces an iteration. However, nested patterns are only used to return a true/false decision. For instance, in class `Pair<X,Y>`, the nested pattern iterates over all the methods in `Y` matching the pattern, but the iteration only serves to verify whether a matching method exists, not to produce different code for each matching method. Furthermore, multiple nested patterns are all nested at the same level, forming a conjunction of their conditions.

Nested patterns may use pattern variables that are not bound by any primary

pattern. However, there are restrictions as to how variables bound only by nested patterns can be used in code introduced by the reflective block (i.e., the reflective declaration). Pattern variables bound by only negative nested patterns cannot be used in the reflective declaration at all. For instance, the pattern "`no public R restore()`", above, binds type variable `R`. Since `R` only appears in a negative nested pattern, it cannot be used in the declaration of `restore()`. Intuitively, a variable in a negative nested pattern is never bound to any concrete type or identifier—no match can exist for the negative nested condition to be satisfied. Clearly, an unbound variable cannot be used in declarations.

Pattern variables that are bound by positive nested patterns, however, can be used in the reflective declaration, if we can determine that exactly one element can be matched by the nested pattern. This is the case only if all uniquely identifying parts of the nested pattern use either constants, or pattern variables bound by the primary pattern. The uniquely identifying parts of a method pattern are the method's name and argument types, and the uniquely identifying part of a field pattern is the field's name. For example, in `Pair<X,Y>`, the positive nested pattern "`some RY m(A) : Y.methods`" uses `m` and `A` in its uniquely identifying parts. Both pattern variables are bound by the primary pattern. Thus, we can use `RY` in the reflective declaration, even though it only appears in the nested pattern.

It may not be immediately obvious how `if` and `errorif` relate to nested patterns. However, the type system machinery that enables `if` and `errorif` is precisely that of nested patterns. The patterns used as `if` and `errorif` conditions are regular nested patterns (with `some` and `no`) with the same semantics and conditions (e.g., limitations on when bound variables can appear in a reflective declaration). Indeed, even our actual implementation of the static `if` statement translates it intermediately into a static `for` loop with a special "unit" value for the primary pattern condition.

We do not allow the nesting of primary patterns—i.e., it is not legal to have nested static `for` loops. However, `if` and `errorif` declarations and statements can be freely nested within the scopes of one another, or within the scope of a static `for` loop.

5. APPLICATIONS USING NESTED PATTERNS

We next show two real world applications, re-implemented concisely and safely with nested patterns.

5.1 DSTM2

DSTM2 [Herlihy et al. 2006] is a Java library implementing object-based software transactional memory. It provides a number of "transactional factories" that take as input a sequential class, and generate a transactional class. Each factory supports a different transactional policy. The strength of DSTM2 is in its flexibility. Users can mix and match policies for objects, or define new "factories" implementing their own transactional policies.

In order to automatically generate transactional classes, DSTM2 factory classes use a combination of Java reflection, bytecode engineering with BCEL [Apache Software Foundation], and anonymous class definitions. For instance, for any input Java class, DSTM2 uses the Java reflection API to retrieve all fields annotated with `@atomic`, and generates appropriate "getter" and "setter" methods for these fields

by injecting the bytecode representation of these methods into the input class file. The information needed for the generation of these methods is purely static and structural. The authors of DSTM2 had to employ low-level run-time techniques only because the Java language does not offer enough support for compile-time transformation of classes. MorphJ, however, is a good fit for this task.

In our re-implementation of DSTM2's factories and supporting classes, 1,484 (non-commented, non-blank) lines of Java code are replaced with 576 lines of MorphJ code. For example, we replaced DSTM2's `factory.shadow.Adaptor<X>` and `factory.shadow.RecoverableFactory<X>` with the MorphJ class `Recoverable<X>` in Figure 6.

For each field of `X`, `Recoverable<X>` creates a shadow field, as well as getter and setter methods that acquire a lock from a transactional manager first, perform the read or write, and then resolve conflicts before returning. Furthermore, it creates `backup()` and `restore()` methods to backup and restore fields to and from their shadow fields.

The advantage of the MorphJ implementation is two-fold. First, `Recoverable<X>` is guaranteed to never declare conflicting declarations. For example, `shadow#f` is only declared if this field does not already exist in `X`, and `backup()` is only declared if such a method does not already exist in `X`. Implementations using reflection and bytecode engineering enjoy no such guarantees, and must instead rely on thorough testing to discover potential bugs.

Additionally, class `Recoverable<X>` is easier to write and understand. For example, the code for generating a `backup()` method in DSTM2's `RecoverableFactory<X>` is illustrated in Figure 7. We invite the reader to compare the `backup()` method declaration in Figure 6 (lines 28-32) to the code in Figure 7.

5.2 Default Implementations for Interface Methods

Consider again the "default implementation" of interface methods discussed in Section 3.2. What if we want a more advanced version that will not provide do-nothing implementations for *all* methods of an interface, but only for those that are not already implemented in a class? There have been mechanisms proposed [Huang and Smaragdakis 2006; Mohnen 2002] that specifically target this problem through the use of new keywords (or Java annotations). Such mechanisms either have no guarantee for the well-typedness of generated code [Huang and Smaragdakis 2006], or require extensions to the Java type system [Mohnen 2002]. In contrast, we can replace these language extensions with a MorphJ generic class that is guaranteed to always produce well-typed code. This turns out to be a surprisingly interesting example, requiring careful reasoning about nested patterns in order to type-check modularly. Figure 8 shows a slightly simplified version of the MorphJ solution to this problem. (For conciseness, we elide the declarations dealing with `void`- or primitive-type-returning methods, which roughly double the code.)

Class `DefaultImplementation<X,I>` copies all methods of type `X` that either correctly implement methods in interface `I`, or are guaranteed to not conflict with methods in `I`. For methods in `I` that have no counterpart in `X`, a default implementation is provided. Methods in `X` that conflict with methods in `I` (same argument types, different return types) are ignored. The code for `DefaultImplementation<X,I>` demonstrates the power of nested patterns, both in terms of expressiveness, and in

```
1  @atomic public class Recoverable<class X> extends X {
2    // for each atomic field of X, declare a shadow field.
3    <F>[f]for(@atomic F f: X.fields; no shadow#f: X.fields)
4    F shadow#f;
5
6    // for each field of X, declare a getter.
7    <F>[f]for(@atomic F f: X.fields; no get#f(): X.methods)
8    public F get#f () {
9      Transaction me  = Thread.getTransaction();
10     Transaction other = null;
11     while (true) {
12       synchronized (this) {
13         other = openRead(me);
14         if (other == null) { return f; }
15       }
16       ... // code resolving conflict between me and other.
17     }
18   }
19   // for each field of X, declare a setter
20   <F>[f]for(@atomic F f : X.fields; no set#f(F) : X.methods)
21   public void set#f ( F val ) {
22     ... // code to open transaction.
23     f = val;
24     ... // code resolving conflict.
25   }
26
27   // create backup method
28   <R>if ( no R backup() : X.methods )
29   public void backup() {
30     <F>[f] for (@atomic F f : X.fields)
31     shadow#f = f;
32   }
33   // create recover method
34   <R>if ( no R recover() : X.methods )
35   public void recover() {
36     // restore field values from shadow fields.
37     <F>[f] for ( @atomic F f : X.fields )
38     f = shadow#f;
39   }
40 }
```

Fig. 6. A recoverable transactional class in MorphJ.

```
public class RecoverableFactory<X> extends BaseFactory<X> {
   ...
  public void createBackup() {
    InstructionList il = new InstructionList();
    MethodGen method =
      new MethodGen(ACC_PUBLIC, Type.VOID, Type.NO_ARGS,
                    new String[] { }, "backup", className, il, _cp);

    for (Property p : properties) {
      InstructionHandle ih_0 =
        il.append(_factory.createLoad(Type.OBJECT, 0));
      il.append(_factory.createLoad(Type.OBJECT, 0));
      il.append(_factory.createFieldAccess(className, p.name,
                                           p.type, Constants.GETFIELD));
      il.append(_factory.createFieldAccess(className, p.name + "$",
                                           p.type, Constants.PUTFIELD));
    }

    InstructionHandle ih_24 = il.append(_factory.createReturn(Type.VOID));
    method.setMaxStack();
    method.setMaxLocals();
    _cg.addMethod(method.getMethod());
    il.dispose();
  }
}
```

Fig. 7. DSTM2 code for creating a method `backup()`.

terms of type safety. The application naturally calls for different handling of methods in a type, based on the existence of methods in another type. Furthermore, these declarations are guaranteed to be unique, and their uniqueness is crucially based on nested patterns.

6. TYPE-CHECKING MORPHJ: A CASUAL DISCUSSION

Higher variability always introduces complexity in type systems. For instance, polymorphic types require more sophisticated type systems than monomorphic types, because polymorphic types can reference type "variables", whose exact values are unknown at the definition site of the polymorphic code. In MorphJ, in addition to type variables, there are also *name* variables—declarations and references can use names reflectively retrieved from type variables. Thus, the exact values of these names are not known when writing a generic class. Yet, the author of the generic class needs to have some confidence that his/her code will work correctly with any parameterization in its intended domain. The job of MorphJ's type system is to ensure that generic code does not introduce static errors, for *any* type parameter that satisfies the author's stated assumptions.

There are two main challenges in type-checking a MorphJ program: 1) how do we determine that declarations made with name variables are unique, i.e., there

```
class DefaultImplementation<X,interface I> implements I {
  X x;
  DefaultImplementation(X x) { this.x = x; }

  // for all methods in I, if the same method does
  // not appear in X, provide default implementation.
  <R extends Object,A*>[m]for( R m (A) : I.methods ;
                               no R m (A) : X.methods )
  R m (A a) { return null; }

  // for all methods in X that *do* correctly override
  // methods in I, we need to copy them.
  <R,A*>[m]for(   R m (A) : I.methods ;
              some R m (A) : X.methods )
  R m (A a) { return x.m(a); }

  // for all methods in X, such that there is no method
  // in I with the same name and arguments, copy method.
  <R,A*>[m]for( R m (A) : X.methods;
              no m (A) : I.methods)
  R m (A a) { return x.m(a); }
}
```

Fig. 8. A MorphJ generic class providing default implementations of methods in any interface `I`.

are no naming conflicts, when we do not know the exact identifiers these name variables could represent, and 2) how do we determine that references always refer to declared members and are well-typed, when we know neither the exact names of the members referenced, or the exact names of the members declared. The key idea is to reason at the level of a set of syntactic elements, instead of a single element. We define the *generation range* of a reflective declaration (or reference) as the set of declarations (resp. references) it produces. (A regular, non-reflective declaration or reference has a one-element generation range.) We shorten the term "generation range" to just *range*, just as we did with the "reflective range", when it is clear from context whether we refer to the elements matched by a pattern or to the elements generated by a reflective block. Determining the lack of conflicts between two declarations then reduces to determining whether their ranges are *disjoint*. Determining whether a reference is valid reduces to determining *containment*: Are all entities in the reference range contained, i.e., have corresponding entities, within the declaration range?

In this section, we present informally, through examples the main problems and insights for checking declaration uniqueness and reference validity in MorphJ programs. We focus on declarations and references made by reflecting over type variables: Reflecting over non-variable types is simply syntactic sugar for manually inlining the declarations. We further focus on the rules for type-checking methods—rules for fields are a trivial adaptation of those for methods.

6.1 Uniqueness of Declarations

We use range disjointness to check whether declarations are unique. In the case of method declarations, uniqueness means two methods within the same class (including inherited methods) cannot have the same name and argument types.

6.1.1 *Internally Well-defined Range.* A simple property to establish is the uniqueness of declarations introduced by the same reflective iteration block.

Simple case. Consider a simple MorphJ class:

```
class CopyMethods<X> {
  <R,A*>[m] for( R m (A) : X.methods ; nestedConditions )
  R m (A a) { ... }
}
```

`CopyMethods<X>`'s methods are declared within one reflective block, which iterates over all the methods of type parameter `X`. For each method returning a non-`void` type that also satisfies *`nestedConditions`*, a method with the same signature is declared for `CopyMethods<X>`.

How do we guarantee that, given any `X`, `CopyMethods<X>` has unique method declarations (i.e., each method is uniquely identified by its ⟨name, argument types⟩ tuple)? Observe that `X` can only be instantiated with another well-formed type, and all well-formed types have unique method declarations. Thus, if a type merely copies the method signatures of another well-formed type, as `CopyMethods<X>` does, it is guaranteed to have unique method signatures, as well. The same principle also applies to reflective field declarations. Nested conditions only serve to remove more methods from the reflective range. Thus, regardless of the specific *`nestedConditions`*, the above declaration is always legal.

It is important that reflective declarations copy *all* the uniquely identifying parts of a method or field. For example, the uniquely identifying parts of a method are its name *together with* its argument types. Thus, a reflective method declaration that only copies either name or argument types would not be well-typed. For example:

```
class CopyMethodsWrong<X> {
  <R,A*>[m] for( R m (A) : X.methods )
  R m () { ... }
}
```

The reflective declaration in `CopyMethodsWrong<X>` only copies the return type and the name of the methods of a well-formed type. This would cause an error if instantiated with a type with an overloaded method:

```
class Overloaded {
  int bar (int a);
  int bar (String s);
}
```

`CopyMethodsWrong<Overloaded>` would have two methods, both named `bar`, taking no arguments.

There is no way to express nested conditions that would filter out methods defined using overloaded method names. Such a nested condition would need to explicitly state that there are no methods in `X` with the same name, but *different* arguments

than the methods matched by the primary pattern. MorphJ explicitly does not allow such inequality conditions.

Beyond Copy and Paste. Morphing of classes and interfaces is not restricted to copying the members of other types. Matched type and name variables can be used freely in reflective declarations and statements. For example:

```
class ChangeArgType<X> {
  <R,A extends Object>[m] for ( R m (A) : X.methods ; nestedConditions )
  R m ( List<A> a ) { /* do for all elements */ ... }
}
```

In `ChangeArgType<X>`, for each method of `X` that takes one non-primitive type argument `A` and returns a non-`void` type `R`, a method with the same name and return type is declared. However, instead of taking the same argument type, this method takes a `List` instantiated with the original argument type. Even though `ChangeArgType<X>` does not copy `X`'s method signatures exactly, we can still guarantee that all methods of `ChangeArgType<X>` have unique signatures, no matter what `X` is. The key is that a reflective declaration can manipulate the uniquely identifying parts of a method, (i.e., name and argument types), by using them in type (or name) compositions, as long as these parts remain in the uniquely identifying parts of the new declaration. The following is an example of an *illegal* manipulation of types:

```
class IllegalChange<X> {
  <R,A>[m] for ( R m (A) : X.methods ; nestedConditions )
  A m ( R a ) { ... }
}
```

In the above example, the uniquely identifying part of `X`'s method is no longer the uniquely identifying part of `IllegalChange<X>`'s method: the argument types of `X`'s method is no longer part of the argument types of `IllegalChange<X>`'s method. Using class `Overloaded` defined above, `IllegalChange<Overloaded>` will cause an error in the generated code. Again, no nested conditions can prevent this type of declaration conflict.

6.1.2 *Uniqueness Across Ranges.* When there are multiple reflective blocks in the same type declaration, we need to guarantee that the declarations in one block do not conflict with the declarations in another block. For two reflective method declarations, their uniqueness means that the generation ranges of their ⟨name, argument types⟩ tuples cannot overlap. This can be determined by a *two-way unification* of the two declarations. In a two-way unification, pattern variables from *both* reflective blocks are unification variables.

Let us start with a simple example. Consider the following class:

```
1 class DisjointDecs<X> {
2   <R>[m] for(R m (int) : X.methods; nestedConditions1 )
3   R m (int i) { ... }
4
5   <S>[n] for(S n (int) : X.methods; nestedConditions2 )
6   S n (int i, String s) { ... }
7 }
```

It is easy to see that the declarations on lines 3 and 6 cannot overlap for any `X`. There is no unification to make the two signatures have the same ⟨name, argument types⟩ tuple, because there is simply no way to unify {`int`} with {`int`,`String`}.

The absence of a possible unification is the most straightforward way to prove that two reflective ranges do not overlap. However, even when two method signatures do unify, if we can prove that conditions causing the overlap are infeasible, then the declarations are still unique. An overlap is infeasible if the unification mapping producing the overlap, when applied to the primary and nested patterns, produces mutually exclusive conditions. Note that a non-empty primary pattern constitutes a positive condition that states that *some* element exists in this range.

Consider the following class:

```
1  class StillUnique<X> {
2    <A1>[m]for( String m (A1) : X.methods ; nestedConditions1 )
3    void m (A1 a) { ... }
4
5    <A2>[n]for( int n (A2)    : X.methods ; nestedConditions2 )
6    void n (A2 a) { ... }
7  }
```

The declared signatures on lines 3 and 6 unify with the mapping {`m`↦`n`, `A1`↦`A2`}. Applying this mapping to the primary patterns on lines 2 and 5, we get "`String n (A2) : X.methods`", and "`int n (A2) : X.methods`". Methods matched by these patterns can cause conflicting declarations. However, having at least one method in both of these ranges means that there need to be two methods in `X` with the same name and argument types, but different return types. This directly contradicts the fact that `X` is a well-formed type. Thus, this unification mapping produces mutually exclusive conditions between the two primary pattern conditions, and there are no elements that would make the mapping possible. These declarations are thus still disjoint.

To generalize the rules for range disjointness, we first define conditions under which two pattern conditions are mutually exclusive. Let $\langle P_n,T_n\rangle$ denote the range of pattern P_n matching over the methods of type T_n, let + prefix a positive pattern condition, and − prefix a negative condition. There are two conditions for mutual exclusivity:

- $+\langle P_n,T_n\rangle$ and $+\langle P'_n,S_n\rangle$ are mutually exclusive if T_n is a subtype of S_n or vice versa, and any unifying method name and argument types for P_n, P'_n have incompatible[3] return types.
- $+\langle P_n,T_n\rangle$ and $-\langle P'_n,S_n\rangle$ are mutually exclusive if $\langle P'_n,S_n\rangle$ contains $\langle P_n,T_n\rangle$.

We discuss range containment in detail in Section 6.2. Intuitively, $\langle P'_n,S_n\rangle$ contains $\langle P_n,T_n\rangle$ if all members in the range of $\langle P_n,T_n\rangle$ are in the range of $\langle P'_n,S_n\rangle$. Naturally, having something in a larger range conflicts with having nothing in a smaller range.

[3]The rule refers to "incompatible" instead of just "different" types because, in case T_n is a proper subtype of S_n, it can contain an overriding method with a different but covariant return type, which would result in a conflicting declaration. We try to be precise in our statements and examples but mostly ignore such tedious corner cases in our informal discussion.

Given two method signatures whose uniquely identifying parts do unify (which suggests a conflict is possible), we first apply the unification mapping to their enclosing primary and nested patterns. Next, we determine whether the two primary pattern conditions are mutually exclusive using the rules above. If the primary pattern conditions are not mutually exclusive, that is, the patterns unify, we apply the unification mapping to all patterns again. We then determine whether any pair of conditions are mutually exclusive. If there is at least one pair of mutually exclusive conditions, the two method generation ranges must be disjoint.

We saw the first rule's use in our `StillUnique<X>` example. The second rule is what establishes that ranges are disjoint because the negative nested pattern of one contains the primary pattern of the other. Consider the following example:

```
1  public class UnionOfStatic<X,Y> {
2    <A*>[m] for( static void m (A) : X.methods; nestedConditions)
3    public static void m(A args) { X.m(args); }
4
5    <B*>[n] for( static void n (B)      : Y.methods ;
6              no static void n (int, B) : X.methods )
7    public static int n(int count, B args) {
8      for (int i = 0; i < count; i++) Y.n(args);
9      return count;
10   }
11 }
```

The two method declarations on lines 3 and 7 have signatures that can be unified with the mapping { A↦{int,B}, m↦n}. Applying this substitution to the primary pattern on line 2 yields "`static void n(int,B) : X.methods`". The range of methods matched by this pattern is contained in the range of methods matched by the nested pattern on line 6. That is, the conflict would fall under the range of the negative nested pattern "`no static void n(int,B) : X.methods`". Thus, the conflict is not possible and the two method declarations are unique for all `X` and `Y`.

Reflective and Regular Methods Together. Declaration conflicts can also occur when a class has both regular and reflectively declared members. For example, in the following class declaration, we cannot guarantee that the methods declared in the reflective block do not conflict with method `int foo()`.

```
class Foo<X> {
  int foo () { ... }

  <R,A*>[m]for ( R m (A) : X.methods ; nestedConditions )
  R m (A a) { ... }
}
```

Just as in the case of multiple iterators, the main issue is establishing the disjointness of declaration ranges, with the regular methods acting as a constant declaration range. In this case, a simple way is to use a nested condition to ensure that there is no method named `foo` in `X`: `no foo(S) : X.methods`, where `S` is declared as pattern type variable `S*`. This nested condition is mutually exclusive with the primary pattern condition—when unified with the regular method signature `int foo()`, no

method in the range of the substituted primary pattern can possibly satisfy the nested condition, and thus no method will be declared at all through this reflective block.

Using Static Prefixes. In general, we can guarantee the uniqueness of declarations across reflective blocks by proving either type signature or name uniqueness. A general way to establish the uniqueness of declarations is by using unique static prefixes on names. (For static prefixes to be uniquely identifying, they must not be prefixes of each other.) For instance, the following class is guaranteed to always produce unique method declarations:

```
class Manipulation<X> {
  <R>[m] for ( R m (List<X>) : X.methods )
  R list#m (List<X> a) { ... }

  <R>[m] for ( R m (X) : X.methods )
  R nolist#m (List<X> a) { ... }
}
```

Proper Method Overriding and Mixins. Proper overriding means that a subtype should not declare a method with the same name and arguments as a method in a supertype, but a non-covariant return type. Ensuring proper method overriding is a special case of declaration range disjointness: if two methods' ⟨name, argument types⟩ tuples are not unique, they are still well-typed declarations if we can establish that the overriding method's return type is a subtype of the overriden return type.

One case that deserves some discussion is that of a type variable used as a supertype. (In case the type is a class, it is implicitly assumed to be non-final.) This is sometimes called a *mixin* pattern [Bracha and Cook 1990; Smaragdakis and Batory 1998]. Since the supertype could potentially be any type, we have no way of knowing its declarations. For instance, the following class is unsafe and will trigger a type error, as there is no guarantee that the superclass does not already contain an incompatible method `foo`.

```
class C<class T> extends T {
  int foo () { ... }
}
```

Static prefixes are similarly insufficient to guarantee that subtype methods do not conflict with supertype methods. As a result, there are two legal ways to declare (non-empty) mixins in MorphJ.

First, the subclass may contain *no* members other than reflective iterators over its supertype that declare overriding versions for (some subset of) the supertype's methods. For instance, the following is a legal MorphJ class:

```
class C<class T> extends T {
  <R,A>[m] for (R m (A) : T.methods)
  R m (A a) { ... }
}
```

The class correctly overrides some of its superclass's methods (those accepting and returning one argument).

Alternatively, the subclass may use proper nested conditions to eliminate possibilities of illegal overriding. For instance:

```
class C<class T> extends T {
  if ( no foo () : T.methods )
  int foo () { ... }
}
```

6.2 Validity of References

Another challenge of modular type checking for a morphing language is to ensure the validity of references. We use the term "validity" to refer to the property that a referenced entity has a definition, and its use is well-typed.

6.2.1 *Reference within the Same Range.* Let us take another look at the class `Logging<X>` from Section 1:

```
1 class Logging<class X> extends X {
2   <R,Y*>[meth]for(public R meth (Y) : X.methods)
3   public R meth (Y a) {
4     R r = super.meth(a);
5     System.out.println("Returned: " + r);
6     return r;
7   }
8 }
```

How do we know that the method invocation "`super.meth(a)`" (line 4) is valid? Notice that the generation range of `meth` (i.e., all the identifiers it could expand to) is exactly the names of methods matched by the primary pattern on line 2: all non-`void` methods of `X`. This range is certainly contained by the set of all methods declared for `X`. Thus, we know method `meth` exists, no matter what `X` is. Furthermore, how do we know we are invoking `meth` with the right arguments? The type of `a` is `Y`: exactly the argument type `m` of `X` is expecting.

6.2.2 *Reference Across Ranges.* Things get a bit more complex when a name variable bound in one reflective block references a method declared in a different reflective block. Consider the following class, which logs the arguments of methods accepting strings, before calling `Logging` to log the return value.

```
1 class LogStringArg<class Y> {
2   Logging<Y> loggedY;
3
4    <T>[n] for ( public T n(String) : Y.methods )
5    public T n (String s) {
6        System.out.println("arg: " + s);
7        return loggedY.n(s);
8    }
9 }
```

How do we know that `loggedY.n(s)` (line 7) is a valid reference, when the methods of `loggedY` are defined in a different class and a different reflective block? The key is to determine that the range of `n` is contained by the range of method names

in `Logging<Y>`. This is to say that the generation range of `n`'s enclosing reflective block should be contained by the generation range of `Logging<Y>`'s declaration reflective block. In general, this containment check has two components (beyond the regular matching of names and signatures): *the reflection set of the containing range (i.e., the type it is defined over) should be richer than (i.e., a subtype of) the reflection set of the contained range, and the pattern of the containing range should be more general than that of the contained range.* Observe that the declaration block of `Logging<Y>` is defined over methods of `Y` (after substituting `Y` for `X`), as is the reflective block enclosing `n`. Secondly, the pattern for the declaration block of `Logging<Y>` is more general than the pattern for the reflective block enclosing `n`: the former matches all non-`void` methods, and the latter matches all non-`void` methods taking exactly one `String` argument. Thus, any method that is matched by the reference reflective block's pattern is matched by the declaration reflective block's pattern, regardless of what `Y` is. Consequently, there is always a method `n` in `Logging<Y>`.

Whether a pattern is more general than another can be systematically determined by finding a *one-way unification* from the more general pattern to the more restricted one. In a one-way unification, only pattern variables declared for the more general pattern are used as unification variables. All other pattern variables are considered constants. In this example, we can unify "`public R m(A)`" to "`public T n(String)`" using the mapping {`R`↦`T`, `m`↦`n`, `A`↦{`String`}}.

We also use this unification mapping in determining whether `n` is invoked with the right argument types. We apply the mapping to the method declaration in `Logging<Y>`, and get the declared signature "`public T n(String)`". Since `s` has the type `String`, the invocation is clearly correct. Furthermore, we can check that the result of the invocation is of type `T`, which is precisely the expected return type of the method enclosing "`loggedY.n(s)`".

For a case with nested patterns, consider the following class:

```
1  class VoidPair<X,Y> {
2    X x; Y y; .../// constructor to initialize x and y.
3
4    <A*>[m]for (  public void m(A) : X.methods ;
5              some public void m(A) : Y.methods )
6    public void m (A a) {   x.m(a);  y.m(a);   }
7  }
```

`VoidPair<X,Y>` declares a method for every `void` method that `X` and `Y` share in name and argument types, and invokes that method on `x` and `y`. Using the reference rules described previously, we know that `x.m(a)` is a valid reference. Furthermore, because the pattern variables used in the positive nested pattern on line 5 are all bound by the primary pattern, we know that if the nested condition is satisfied, there is exactly one element in the range of the nested pattern, so the call `y.m(a)` is unambiguous. Since the types also match, `y.m(a)` is a valid reference, as well.

We can abstract away from these examples and consider the general case of a reference made in one reflective block, to declarations made in another reflective block, when both blocks have nested patterns. Let R_d and R_r denote the ranges for the reflective blocks of the declaration and the reference, respectively. There

are two sufficient conditions to determine that R_r is contained in R_d. First, the primary range of R_r (i.e., the range of the primary pattern defining R_r) must be contained in the primary range of R_d, using the same check as before (one-way unification of the patterns that define the ranges). Second, for all methods that are in the primary range of R_r (and thus also in the primary range of R_d), if the method satisfies the nested conditions of R_r, it should also satisfy the nested conditions of R_d. That is to say, the nested conditions of R_r should be stronger, and imply the nested conditions of R_d.

Determining that one nested condition implies another can be reduced to another single range containment (i.e., one-way pattern unification) check. More precisely, using the same notation as before (recall that T_r and T_d are the types on whose methods the ranges iterate), we have two ways of determining that one nested condition implies another:

- $+\langle N_r,T_r\rangle$ implies $+\langle N_d,T_d\rangle$ if $\langle N_d,T_d\rangle$ contains $\langle N_r,T_r\rangle$ (i.e., T_d is a subtype of T_r and the one-way unification of patterns succeeds).
- $-\langle N_r,T_r\rangle$ implies $-\langle N_d,T_d\rangle$ if $\langle N_r,T_r\rangle$ contains $\langle N_d,T_d\rangle$.

Intuitively, $+\langle N_r,T_r\rangle$ is satisfied if there is at least one element in $\langle N_r,T_r\rangle$. Then there is certainly at least one element in a larger range, as well. Thus, $+\langle N_d,T_d\rangle$ should be satisfied. Similar reasoning applies for the implication between two negative conditions.

To be more concrete, consider the following class:

```
 8  class CallVoidsWithString<T,S> {
 9    VoidPair<T,S> voidPair;
10    ... // constructor to initialize voidPair
11    [n]for (  public void n(String) : T.methods ;
12         some public void n(String) : S.methods )
13    public void n (String s) { voidPair.n(s); }
14  }
```

For every `void` method taking one `String` argument that `T` and `S` have in common, `CallVoidsWithString<T,S>` declares a method with the same signature, and invokes a method with the same name on `voidPair`, of type `VoidPair<T,S>`. This reference is valid if the range of the reflective block on lines 11-12 is contained by the range of the declaration reflective block (lines 4-5 in the definition of `VoidPair`).

The range of the primary pattern on line 11 is contained by the range of the declaration's primary pattern (line 4), by the one-way unification mapping {`m`↦`n`, `A`↦{`String`}}.

To check whether the nested pattern on line 5 contains the nested pattern on line 12, note that we first apply the unification mappings obtained from unifying the primary patterns—we only want to determine this containment relationship for those methods that belong in the range of both primary patterns. In our example, after applying the unification mapping to the positive nested pattern on line 5 (and also substituting `S` for `Y`), we have "`public void n(String) : S.methods`". This clearly contains "`public void n(String) : S.methods`" on line 12.

These two conditions guarantee us that reference `voidPair.n(s)` is always a valid one. It is easy to check that this is indeed the case.

The above approach generalizes to an arbitrary number of nested conditions: Each nested condition in the declaration range must be implied by at least one nested condition in the reference range. A range with no nested patterns is equivalent to a range with a positive nested pattern that contains everything, or a negative nested pattern that is contained by everything. The case where there are only nested patterns (i.e., `if` and `errorif` statements) can be reduced to a range with a special primary pattern value that contains only itself and is contained only by itself.

7. FORMALIZATION

We formalize the main features of MorphJ and prove type soundness through a formalism, FMJ, based on FGJ [Igarashi et al. 2001], with differences (other than the simple addition of our extra environment, Λ) highlighted in grey. Figures in which all rules are new to FMJ (Figures 12,14,15) are not highlighted at all, for better readability.

7.1 Syntax

The syntax of FMJ is presented in Figure 9. We adopt many of the notational conventions of FGJ: `C`,`D` denote constant class names; `X`,`Y`,`Z` denote type variables; `N`,`P`,`Q`,`R` denote non-variable types (which may be constructed from type variables); `S`,`T`,`U`,`V`,`W` denote types; `f` denotes field names; `m` denotes non-variable method names; `x`,`y` denote argument names. Notations new to FMJ are: η denotes a variable method name; `n` denotes either variable or non-variable names; o denotes a nested condition operator (either + or -, for the keywords `some` or `no`, respectively).

T	::=	X | N
N	::=	C<$\overline{\texttt{T}}$>
CL	::=	class C<$\overline{\texttt{X}}$◁$\overline{\texttt{N}}$>◁ N {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\texttt{M}}$}
	|	class C<$\overline{\texttt{X}}$◁$\overline{\texttt{N}}$>◁ T {$\overline{\texttt{T}}$ $\overline{\texttt{f}}$; $\overline{\mathfrak{M}}$}
M	::=	T m ($\overline{\texttt{T}}$ $\overline{\texttt{x}}$) {↑e;}
$\mathfrak{M}$	::=	<$\overline{\texttt{Y}}$◁$\overline{\texttt{P}}$>for($\mathbb{M}_p$;$o\mathbb{M}_n$) U η ($\overline{\texttt{U}}$ $\overline{\texttt{x}}$) {↑e;}
o	::=	+ | -
$\mathbb{M}$	::=	V η ($\overline{\texttt{V}}$):X.methods
e	::=	x | e.f | e.n ($\overline{\texttt{e}}$) | new C<$\overline{\texttt{T}}$>($\overline{\texttt{e}}$)
n	::=	m | η

Fig. 9. FMJ: Syntax

We use the shorthand $\overline{\texttt{T}}$ for a sequence of types $\texttt{T}_0\texttt{,T}_1\texttt{,...,T}_n$, and $\overline{\texttt{x}}$ for a sequence of unique variables $\texttt{x}_0\texttt{,x}_1\texttt{,...,x}_n$. We use : for sequence concatenation, e.g., $\overline{\texttt{S}}\texttt{:}\overline{\texttt{T}}$ is a sequence that begins with $\overline{\texttt{S}}$, followed by $\overline{\texttt{T}}$. We use $\bullet$ to denote an empty sequence. We use $\in$ to mean "is a member of a sequence" (in addition to set membership). We use ... for values of no particular significance to a rule. ◁ and ↑ are shorthands for the keywords `extends` and `return`, respectively. Note that all classes must declare a superclass, which can be `Object`.

FMJ formalizes some core MorphJ features that are representative of our approach. One simplification is that we allow a single nested pattern and it cannot

use any pattern type or name variables not bound by its primary pattern. We also do not formalize reflecting over a statically known type, or using a constant name in reflective patterns. These are decidedly less interesting cases from a typing perspective. The zero or more length type vectors `T*` are also not formalized. These type vectors are a mere matching convenience. Thus, safety issues regarding their use are covered by non-vector types. We do not formalize reflectively declared fields—their type checking is a strict adaptation of the techniques for checking methods. Lastly, static name prefixes, casting expressions and polymorphic methods are not formalized.

FGJ does not support method overloading, and FMJ inherits this restriction. Thus, a method name alone uniquely identifies a method definition. Furthermore, since we allow no fresh name variables in nested patterns, there can be only one name variable in a pattern, and we use the keyword η for name variables, instead of allowing arbitrary identifiers. A reflective definition must also use this same name (since static prefixes are not allowed and constant names would be illegal due to conflicts). This results in a small simplification over the informal rules, but leaves their essence intact.

A program in FMJ is an $(\mathtt{e}, CT)$ pair, where `e` is an FMJ expression, and CT is the class table. We place some conditions on CT: Every class declaration `class C...` has an entry in CT; `Object` is not in CT. The subtyping relation derived from CT must be acyclic, and the sequence of ancestors of every instantiation type is finite. (The last two properties can be checked with the algorithm of [Allen et al. 2003] in the presence of mixins.)

7.2 Typing Judgments

The main typing rules of FMJ are presented in Figure 10 and 11, with auxiliary definitions presented in Figures 12 and 13. *We recommend reading our text description and referring to the rules as needed in the flow of the text.*

There are three environments used in our typing judgments:

—Δ: Type environment. Maps type variables to their upper bounds.

—Γ: Variable environment. Maps variables (e.g., `x`) to their types.

—Λ: Reflective iteration environment. Λ has the form $\langle R_p, oR_n \rangle$, where R_p is the primary pattern, and oR_n the nested pattern. Recall that o can be + or -.

  —R_p has the form $(\mathtt{T}_1, \mathtt{<}\overline{\mathtt{Y}}\mathtt{\triangleleft}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\rightarrow}\mathtt{U}_0)$. $\mathtt{T}_1$ is the reflective type, over whose methods R_p iterates. $\overline{\mathtt{Y}}$ are pattern type variables, bounded by $\overline{\mathtt{P}}$, and $\overline{\mathtt{U}}\mathtt{\rightarrow}\mathtt{U}_0$ the method pattern.

  —R_n has a similar form: $(\mathtt{T}_2, \overline{\mathtt{V}}\mathtt{\rightarrow}\mathtt{V}_0)$. However, note the lack of pattern type variables, due to the (formalism-only) simplification that the nested pattern not use pattern type variables not already bound in the primary pattern.

  There is no nesting of reflective loops. Thus, Λ contains at most one $\langle R_p, oR_n \rangle$ tuple.

We define two functions (Figure 12) to help us construct the reflective environment as well as the two ranges corresponding to the primary and nested pattern:

—*reflectiveEnv*($\mathfrak{M}$) constructs the Λ corresponding to the reflective declaration $\mathfrak{M}$.

Expression typing:

$$\Delta;\Lambda;\Gamma\vdash \mathtt{x} \in \Gamma(\mathtt{x}) \qquad \text{(T-VAR)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\in\mathtt{T}_0 \quad \Delta\vdash \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0.\mathtt{f}_i\in\mathtt{T}_i} \qquad \text{(T-FIELD)}$$

$$\frac{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0\in\mathtt{T}_0 \quad \Delta;\Lambda;\Gamma\vdash \overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T}_0)=\overline{\mathtt{T}}\to\mathtt{T} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}}{\Delta;\Lambda;\Gamma\vdash \mathtt{e}_0.\mathtt{n}(\overline{\mathtt{e}})\in\mathtt{T}} \qquad \text{(T-INVK)}$$

$$\frac{\Delta\vdash \mathtt{C}\langle\overline{\mathtt{T}}\rangle\ \mathit{ok} \quad \Delta\vdash \mathit{fields}(\mathtt{C}\langle\overline{\mathtt{T}}\rangle)=\overline{\mathtt{U}}\ \overline{\mathtt{f}} \quad \Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}}{\Delta;\Lambda;\Gamma\vdash \mathtt{new}\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})\in\mathtt{C}\langle\overline{\mathtt{T}}\rangle} \qquad \text{(T-NEW)}$$

Method typing:

$$\frac{\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \quad \Gamma=\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}},\mathtt{this}\mapsto\mathtt{C}\langle\overline{\mathtt{X}}\rangle \quad \Lambda=\emptyset \quad \Delta\vdash\overline{\mathtt{T}},\mathtt{T}_0\ \mathit{ok} \quad \Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{S}_0 \quad \Delta\vdash\mathtt{S}_0<:\mathtt{T}_0 \quad CT(\mathtt{C})=\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{N}\ \{\ldots\} \quad \Delta;\Lambda\vdash\mathit{override}(\mathtt{m},\ \mathtt{N},\ \overline{\mathtt{T}}\to\mathtt{T}_0)}{\mathtt{T}_0\ \mathtt{m}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\ \uparrow\mathtt{e}_0;\ \}\ \mathtt{OK\ IN}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle} \qquad \text{(T-METH-S)}$$

$$\frac{\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}} \quad \Gamma=\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}},\mathtt{this}\mapsto\mathtt{C}\langle\overline{\mathtt{X}}\rangle \quad \Delta\vdash\overline{\mathtt{P}},\overline{\mathtt{T}},\mathtt{T}_0,\overline{\mathtt{N}}\ \mathit{ok} \quad \Delta\vdash\mathbb{M}_p,\mathbb{M}_f\ \mathit{ok} \quad R_p=\mathit{range}(\mathbb{M}_p,\langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\rangle) \quad R_n=\mathit{range}(\mathbb{M}_f,\bullet) \quad \Lambda=\langle R_p,oR_n\rangle \quad \Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{S}_0 \quad \Delta\vdash\mathtt{S}_0<:\mathtt{T}_0 \quad CT(\mathtt{C})=\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{T}\ \{\ldots\} \quad \Delta;\Lambda\vdash\mathit{override}(\eta,\ \mathtt{T},\ \overline{\mathtt{T}}\to\mathtt{T}_0)}{\langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\rangle\mathtt{for}(\mathbb{M}_p;o\mathbb{M}_f)\ \mathtt{T}_0\ \eta\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\ \mathtt{OK\ IN}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle} \qquad \text{(T-METH-R)}$$

Class typing:

$$\frac{\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \quad \Delta\vdash\overline{\mathtt{N}},\mathtt{N},\overline{\mathtt{T}}\ \mathit{ok} \quad \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle}{\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{N}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \qquad \text{(T-CLASS-S)}$$

$$\frac{\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \quad \Delta\vdash\overline{\mathtt{N}},\mathtt{T},\overline{\mathtt{T}}\ \mathit{ok} \quad \text{for all } \mathfrak{M}_i\in\overline{\mathfrak{M}},\ \mathfrak{M}_i\ \mathtt{OK\ IN}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle \quad \text{for all } \mathfrak{M}_i,\mathfrak{M}_j\in\overline{\mathfrak{M}},\ \Lambda_i=\mathit{reflectiveEnv}(\mathfrak{M}_i) \quad \Lambda_j=\mathit{reflectiveEnv}(\mathfrak{M}_j) \quad \Delta\vdash\mathit{disjoint}(\Lambda_i,\ \Lambda_j)}{\mathtt{class}\ \mathtt{C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{T}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathfrak{M}}\}\ \mathtt{OK}} \qquad \text{(T-CLASS-R)}$$

Fig. 10. FMJ: Typing Rules

Well-formed types:

$$\Delta \vdash \texttt{Object}\ ok \qquad \text{(WF-OBJECT)}$$

$$\frac{\texttt{X} \in dom(\Delta)}{\Delta \vdash \texttt{X}\ ok} \qquad \text{(WF-VAR)}$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}}\texttt{<}\overline{\texttt{N}}\texttt{>}\triangleleft\ \texttt{T}\ \{\ \ldots\ \}\quad \Delta \vdash \overline{\texttt{T}}\ ok \qquad \Delta \vdash \overline{\texttt{T}} <: [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ ok} \qquad \text{(WF-CLASS)}$$

Well-formed Patterns:

$$\frac{\mathbb{M} = \texttt{U}_0\ \eta\ (\overline{\texttt{U}})\ \texttt{:T.methods} \qquad \Delta \vdash \texttt{U}_0, \overline{\texttt{U}}, \texttt{T}\ ok}{\Delta \vdash \mathbb{M}\ ok} \qquad \text{(WF-PAT)}$$

Fig. 11. FMJ: Well-formed types and patterns

—*range*($\mathbb{M}$,<$\overline{\texttt{Y}}\triangleleft\overline{\texttt{Q}}$>) constructs the R corresponding to the pattern $\mathbb{M}$, where $\overline{\texttt{Y}}$ are the pattern type variables, and bounded by $\overline{\texttt{Q}}$.

We use the $\mapsto$ symbol for mappings in the environments. For example, $\Delta = \texttt{X} \mapsto \texttt{C<}\overline{\texttt{T}}\texttt{>}$ means that $\Delta(\texttt{X}) = \texttt{C<}\overline{\texttt{T}}\texttt{>}$. Every type variable must be bounded by a non-variable type. The function $bound_\Delta(\texttt{T})$ returns the upper bound of type T in Δ. $bound_\Delta(\texttt{N}) = \texttt{N}$, if N is not a type variable. And $bound_\Delta(\texttt{X}) = bound_\Delta(\texttt{S})$, where $\Delta(\texttt{X}) = \texttt{S}$.

In order to keep our type rules manageable, we make two simplifying assumptions. To avoid burdening our rules with renamings, we assume that pattern type variables have globally unique names (i.e., are distinct from pattern type variables in other reflective environments, as well as from non-pattern type variables). We also assume that all pattern type variables introduced by a reflective block are bound (i.e., used) in the corresponding primary pattern. Checking this property is easy and purely syntactic.

The core of this type system is in determining reflective range containment and disjointness. Thus, we begin our discussion with a detailed explanation of the rules for containment and disjointness.

7.2.1 *Containment and Disjointness.* The range of a reflective environment, $\langle R_p, oR_n \rangle$, comprises methods in the primary range R_p, that also satisfy the nested pattern oR_n. The nested pattern $+R_n$ (or $-R_n$) is satisfied if there is at least one method (or no method, resp.) in the range of R_n. We call ranges of R_p and R_n *single ranges*. In this section, we explain the rules for determining the following three relations:

—$\Delta; [\overline{\texttt{W}}/\overline{\texttt{Y}}] \vdash \Lambda \sqsubseteq_\Lambda \Lambda'$. Range of Λ is contained within the range of Λ', under the assumptions of type environment Δ and the unifying type substitutions of $[\overline{\texttt{W}}/\overline{\texttt{Y}}]$.

—$\Delta; [\overline{\texttt{W}}/\overline{\texttt{Y}}] \vdash R_1 \sqsubseteq_R R_2$. Single range R_1 is contained within single range R_2, under the assumptions of Δ and the unifying type substitutions of $[\overline{\texttt{W}}/\overline{\texttt{Y}}]$.

—$\Delta \vdash$ *disjoint*(Λ, Λ'). The range of Λ and Λ' are disjoint under the assumptions of Δ.

Range Construction Functions:

$$\frac{\mathbb{M}{=}\mathtt{U}_0\ \eta\ (\overline{\mathtt{U}})\ :\ \mathtt{X.methods}}{\mathit{range}(\mathbb{M},{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>})\ {=}(\mathtt{X},\ {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U}_0)}$$

$$\frac{\begin{array}{c}\mathfrak{M}{=}{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\mathtt{for}\ (\mathbb{M}_p;o\mathbb{M}_f)\ \mathtt{U}_0\ \eta\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\\ R_p{=}\mathit{range}(\mathbb{M}_p,{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>})\qquad R_n{=}\mathit{range}(\mathbb{M}_f,\bullet)\qquad \Lambda{=}\langle R_p,R_n\rangle\end{array}}{\mathit{reflectiveEnv}(\mathfrak{M})\ {=}\Lambda}$$

Specializing reflective environment:

$$\frac{\begin{array}{c}\Lambda_d{=}\langle R_p,oR_n\rangle\quad R_p{=}(\mathtt{T}_i,\ {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U})\quad R_n{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}{\to}\mathtt{V})\\ \Delta;\Lambda{\vdash}\mathit{mtype}(\mathtt{m},\ \mathtt{T}_i){=}\overline{\mathtt{U}}'{\to}\mathtt{U}'\quad R'_p{=}(\mathtt{T}_i,\ \overline{\mathtt{U}}'{\to}\mathtt{U}')\\ R'_n{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}'{\to}\mathtt{V}'_0)\qquad \Delta;\Lambda{\vdash}\mathit{mtype}(\mathtt{m},\ \mathtt{T}_j){=}\overline{\mathtt{V}}'{\to}\mathtt{V}'_0\\ o{=}\text{-}\quad\text{implies}\quad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]{\vdash}R'_p{\sqsubseteq}_R R_p\qquad [\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{V}{:}\overline{\mathtt{V}}){=}(\mathtt{V}'_0{:}\overline{\mathtt{V}}')\end{array}}{\Delta;\Lambda{\vdash}\mathit{specialize}(\mathtt{m},\ \Lambda_d){=}\langle R'_p,{+}R'_n\rangle}$$

(SP-+)

$$\frac{\begin{array}{c}\Lambda_d{=}\langle R_p,oR_n\rangle\qquad R_p{=}(\mathtt{T}_i,\ {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U})\qquad R_n{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}{\to}\mathtt{V})\\ \Delta;\Lambda{\vdash}\mathit{mtype}(\mathtt{m},\ \mathtt{T}_i){=}\overline{\mathtt{U}}'{\to}\mathtt{U}'\quad R'_p{=}(\mathtt{T}_i,\ \overline{\mathtt{U}}'{\to}\mathtt{U}')\ \ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]{\vdash}R'_p{\sqsubseteq}_R R_p\qquad R'_n{=}[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R_n\\ \left\{\begin{array}{ll}\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\mathtt{T}_j)\ \text{is undefined} & \text{or}\\ \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\mathtt{T}_j)=\overline{\mathtt{V}}'\to\mathtt{V}'_0\quad [\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{V}:\overline{\mathtt{V}})\neq(\mathtt{V}'_0:\overline{\mathtt{V}}')\quad o=-\end{array}\right.\end{array}}{\Delta;\Lambda{\vdash}\mathit{specialize}(\mathtt{m},\ \Lambda_d){=}\langle R'_p,{-}R'_n\rangle}$$

(SP--)

Subtype range validity:

$$\Delta\vdash\mathit{validRange}(\emptyset,\ \mathtt{T})\qquad\text{(VR-NOREFL)}$$

$$\frac{R_p{=}(\mathtt{X},\ \ldots)}{\Delta\vdash\mathit{validRange}(\langle R_p,{-}R_n\rangle,\mathtt{X})}\qquad\text{(VR-VAR)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C}){=}\mathtt{class}\ \mathtt{C}{<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{N}}{>}{\lhd}\mathtt{S}\ \{\ \ldots\ \ \overline{\mathfrak{M}}\}\\ \Delta\vdash\mathit{validRange}(\Lambda,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S})\\ \text{for all}\quad \mathfrak{M}_i{\in}\overline{\mathfrak{M}}\quad \mathfrak{M}_i{=}{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\mathtt{for}(\mathbb{M}_p;o\mathbb{M}_f)\ldots\\ \Lambda'{=}\mathit{reflectiveEnv}(\mathfrak{M}_i)\\ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]{\vdash}\Lambda{\sqsubseteq}_\Lambda\Lambda'\ \ \text{or}\ \ \Delta{\vdash}\mathit{disjoint}(\Lambda',\ \Lambda)\end{array}}{\Delta\vdash\mathit{validRange}(\Lambda,\ \mathtt{C}{<}\overline{\mathtt{T}}{>})}\qquad\text{(VR-CLASS)}$$

Fig. 12. FMJ: Auxiliary definitions.

Single range containment. In determining the containment between two reflective environments, we must first see how containment is determined between two single ranges. Rule SB-R (Figure 14) defines the two conditions for single range containment. First, the reflective type of the larger range, R_2, should be a subtype of R_1's reflective type. It is only meaningful to talk about containment if the reflection set of R_2 (i.e., all methods of the reflective type of R_2, regardless of whether they can be matched by the pattern) can be mapped *onto* the reflection set of R_1.

Method type lookup:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \qquad R_p{=}(\mathtt{X},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-R1)}$$

$$\frac{\Lambda{=}\langle R_p, {+}R_n\rangle \qquad R_n{=}(\mathtt{X},\ \overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}{\Delta;\Lambda\vdash \mathit{mtype}(\eta,\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-R2)}$$

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \qquad R_p{=}(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \qquad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathit{bound}_\Delta(\mathtt{X})){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{X}){=}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0} \quad \text{(MT-VAR-S)}$$

$$\frac{CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd N\ \{...\ \ }\overline{\mathtt{M}}\mathtt{\}} \qquad \overline{\mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e;}\}}\mathtt{\in}\overline{\mathtt{M}}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)} \quad \text{(MT-CLASS-S)}$$

$$\frac{CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd N\ \{...\ \ }\overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m}\notin\overline{\mathtt{M}}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}\mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})} \quad \text{(MT-SUPER-S)}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd T\ \{...\ \ }\overline{\mathfrak{M}}\mathtt{\}} \\ \mathfrak{M}_i{=}\mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>for(}\mathbb{M}_p\mathtt{;}\,o\mathbb{M}_f\mathtt{)\ S}_0\ \eta\ \mathtt{(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{\uparrow e;\}} \\ \mathfrak{M}_i\in\overline{\mathfrak{M}} \qquad \Lambda_d{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ \begin{cases}\Delta;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d){=}\Lambda_r & \text{if } \mathtt{n}=\mathtt{m}\\ \Lambda_r=\Lambda & \text{if } \mathtt{n}=\eta\end{cases} \quad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d \end{array}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\mathtt{\to}\mathtt{S}_0)} \quad \text{(MT-CLASS-R)}$$

$$\frac{\begin{array}{ll} & CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd T\ \{...\ \ }\overline{\mathfrak{M}}\mathtt{\}} \\ \text{for all} & \mathfrak{M}_i\mathtt{\in}\overline{\mathfrak{M}} \qquad \Lambda_d{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ & \begin{cases}\Delta;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d){=}\Lambda_r & \text{if } \mathtt{n}=\mathtt{m}\\ \Lambda_r=\Lambda & \text{if } \mathtt{n}=\eta\end{cases} \\ \text{implies} & \Delta{\vdash}\mathit{disjoint}(\Lambda_r,\ \Lambda_d) \end{array}}{\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}\mathit{mtype}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})} \quad \text{(MT-SUPER-R)}$$

Field lookup:

$$\Delta{\vdash}\mathit{fields}(\mathtt{Object}) = \bullet \quad \text{(FD-OBJ)}$$

$$\frac{CT(\mathtt{C}){=}\mathtt{class\ C}\mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd\ T\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ ...\}} \qquad \Delta{\vdash}\mathit{fields}(\mathit{bound}_\Delta([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})){=}\overline{\mathtt{D}}\ \overline{\mathtt{g}}}{\Delta{\vdash}\mathit{fields}(\mathtt{C}\mathtt{<}\overline{\mathtt{T}}\mathtt{>}){=}\overline{\mathtt{D}}\ \overline{\mathtt{g}},[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}} \quad \text{(FD-CLASS)}$$

Valid method overriding:

$$\frac{\Delta\vdash \mathit{validRange}(\Lambda,\ \mathtt{T}) \qquad \Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T}){=}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0 \text{ implies } (\overline{\mathtt{V}}{=}\overline{\mathtt{U}} \text{ and } \mathtt{V}_0{=}\mathtt{U}_0)}{\Delta;\Lambda\vdash \mathit{override}(\mathtt{n},\ \mathtt{T},\ \overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0)}$$

Fig. 13. FMJ: Method type lookup, overriding and field lookup.

Reflective range containment:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \quad \Lambda'{=}\langle R'_p, o'R'_n\rangle \quad R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_p \sqsubseteq_R R'_p \quad \begin{cases}\Delta;\bullet\vdash R_n \sqsubseteq_R [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n \text{ if } o=o'=+\\ \Delta;\bullet\vdash [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n \sqsubseteq_R R_n \text{ if } o=o'=-\end{cases}}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \Lambda\sqsubseteq_\Lambda\Lambda'} \quad \text{(SB-}\Lambda\text{)}$$

Single range containment:

$$\frac{R_1{=}(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R_2{=}(\mathtt{T}_2,\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta\vdash\overline{\mathtt{P}}\mathtt{,}\overline{\mathtt{Q}}\ \mathtt{OK} \quad \Delta\vdash\mathtt{T}_2\mathtt{<:}\mathtt{T}_1 \quad \Delta'{=}\Delta,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}},\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \quad \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2} \quad \text{(SB-}R\text{)}$$

Reflective range disjointness:

$$\frac{\Lambda{=}\langle R_p, oR_n\rangle \quad \Lambda'{=}\langle R'_p, o'R'_n\rangle \quad \Delta\vdash {+}R_p\otimes{+}R'_p \text{ or } \Delta\vdash {+}R_p\otimes o'R'_n \text{ or } \Delta\vdash {+}R'_p\otimes oR_n \text{ or } \Delta\vdash oR_n\otimes o'R'_n \text{ or } \Delta\vdash o'R_n\otimes oR'_n}{\Delta\vdash \mathit{disjoint}(\Lambda,\ \Lambda')} \quad \text{(DS-}\Lambda\text{)}$$

Mutually exclusion of range conditions:

$$\frac{R_1{=}(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R_2{=}(\mathtt{S},\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta\vdash\mathtt{T}\mathtt{<:}\mathtt{S} \text{ or } \mathtt{S}\mathtt{<:}\mathtt{T} \quad \Delta'{=}\Delta,\overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}},\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}} \quad \overline{\mathtt{Z}}{=}\overline{\mathtt{X}}\mathtt{:}\overline{\mathtt{Y}} \quad \text{for all } \overline{\mathtt{W}},\ \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\overline{\mathtt{U}},\ \overline{\mathtt{V}}) \quad \text{implies} \quad [\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\mathtt{U}_0{\neq}[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\mathtt{V}_0}{\Delta\vdash {+}R_1\ \otimes\ {+}R_2} \quad \text{(ME-1)}$$

$$\frac{R_1{=}(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to}\mathtt{U}_0) \quad R_2{=}(\mathtt{S},\ \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to}\mathtt{V}_0) \quad \Delta'{=}\Delta,\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}},\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}} \quad \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_1\sqsubseteq_R R_2}{\Delta\vdash {+}R_1\ \otimes\ {-}R_2} \quad \text{(ME-2)}$$

Fig. 14. FMJ: Containment and disjointness rules

We determine this relation using subtyping: A subtype's methods can be mapped onto its supertype's methods. Secondly, R_2's pattern should be more general than R_1's pattern. This means that a *one-way* unification exists from the pattern of R_2 to the pattern of R_1, where only the pattern type variables in R_2 are considered variables in the unification process. $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$ are the substitutions that satisfy such one-way unification: $\Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})$.

Rule UNI (Figure 15) describes a standard unification condition with a twist: unifying substitutions (for pattern type variables) must respect the subtyping bounds of the type variables. For example, the substitution [`Y`/`Object`], where $\Delta\vdash$`Y`<:`Number`, does *not* unify `Y` and `Object`, because the bound of `Y` is tighter than `Object`. Depending on whether the unification variables $\overline{\mathtt{Z}}$ appear in $\overline{\mathtt{T}}$, $\overline{\mathtt{S}}$, or both, the ability to unify $\overline{\mathtt{T}}$ and $\overline{\mathtt{S}}$ may mean all types matched by $\overline{\mathtt{S}}$ can be matched by $\overline{\mathtt{T}}$ (one-way unification), vice versa, or that there is some intersection in the types matched by $\overline{\mathtt{T}}$ and $\overline{\mathtt{S}}$ (two-way unification).

Figure 15 also lists a number of rules defining when type `T` is a valid substitution for `S`: $\Delta\vdash \mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{S}$. This is to say that `T` and `S` can match at least some of the same

types, using $\overline{\mathtt{Z}}$ as pattern type variables. The most complex case in the pattern matching rules is PM-PVARS, which defines when there is an intersection in the types matched by two different pattern type variables, $\mathtt{Z}_i$ and $\mathtt{Z}_j$. Recall that if $\mathtt{Z}_i$ is bounded by type $\mathtt{T}_i$, it can match any subtype of $\mathtt{T}_i$; similarly, $\mathtt{Z}_j$ bounded by type $\mathtt{T}_j$ can match any subtype of $\mathtt{T}_j$. Technically, for there to be an intersection in the types $\mathtt{Z}_i$ and $\mathtt{Z}_j$ can match, there must be a type that is a subtype of both $\mathtt{T}_i$ and $\mathtt{T}_j$. Since we are modeling a core subset of Java without multiple inheritance (of interfaces), this type must be a subtype of either $\mathtt{T}_i$ or $\mathtt{T}_j$. This is to say, either $\mathtt{Z}_i$ has a greater upper bound than $\mathtt{Z}_j$ and is capable of matching every type matched by $\mathtt{Z}_j$, or vice versa. PM-PVARS describes this either-or condition by taking one of the type variables up to its bound, and invoking the pattern matching rules on that bound and the remaining type variable.

PM-VAR says that for a pattern matching type variable `Z` to match a type `T`, the upper bound of `Z` must be able to match the upper bound of `T`. If the bounds are the exact same types (PM-REFL), we are done. If they are types constructed from the same generic type, we recursively invoke the pattern matching rules on their respective type parameters (PM-CL). If `T`'s declared upper bound is a strict subtype of the upper bound of `Z`, rule PM-CL-S traces up the declared superclass of `T`'s upper bound, until either PM-REFL or PM-CL can be invoked, or the class `Object` is reached and no more recursion can occur.

Reflective (nested) range containment. SB-Λ (Figure 14) defines the conditions for the range of reflective environment $\Lambda{=}\langle R_p, oR_n\rangle$ to be contained within the range of $\Lambda'{=}\langle R'_p, o'R'_n\rangle$. These conditions reflect precisely the informal rules we presented in the previous section. First, regardless of nested patterns, the *primary* range of Λ should at least be contained within the primary range of Λ'. Second, for every method in R_p that satisfies the nested pattern oR_n, the corresponding method in R'_p should satisfy the nested pattern $o'R'_n$. There are a couple of ways to guarantee oR_n *implies* $o'R'_n$. If $+R_n$ is true, and R_n is contained within R'_n, then $+R'_n$ is also true (i.e., if there is at least one method in R_n, then there is at least one method in a larger range, R'_n). This condition is expressed by $\Delta;\bullet\vdash R_n \sqsubseteq_R [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n$, if $o{=}o'{=}+$. We apply the unifying type substitutions for the primary ranges to the nested range R'_n: In order to properly compare the ranges of R_n and R'_n, we need to restrict R'_n to what it can be for the methods that are matched by both R_p and R'_p. Note that we are using an empty sequence of type substitutions ($\bullet$) in determining that R_n is contained within $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_n$. This is because nested patterns do not have pattern type variables of their own, and pattern type variables from the primary pattern are treated as constants in the nested patterns. Similarly, if $\text{-}R_n$ is true, and R_n contains R'_n, $\text{-}R'_n$ is also true.

Reflective range disjointness. Disjointness of reflective ranges is defined by rule DS-Λ (Figure 14). DS-Λ says that for two reflective ranges to be disjoint, we must be able to find one pair of mutually exclusive ($\otimes$) pattern conditions—this includes the implicit ($+$) conditions stated by the primary patterns. There are two rules for pattern condition mutual exclusion: ME-1 and ME-2, defined in Figure 14.

ME-1 says that two $+$ pattern conditions are mutually exclusive if the patterns reflect over types with an inheritance relationship, and the patterns' argument types

unify, whereas their return types do not. This means the two patterns are matching over methods of the same argument types, but different return types. For both of these pattern conditions to be simultaneously satisfied, there need to be at least two methods with the same argument types and different return types, declared by the same class or a subclass. This violates the correct method overriding rule of well-formed types. Since these patterns only reflect over well-formed types, these conditions can never be both satisfied.

ME-2 says that $+R_1$ and $-R_2$ are mutually exclusive, if R_1 is contained within R_2. If there is no method in the larger range, then certainly there cannot be at least one method in a smaller range. And vice versa. Thus, the two conditions can never be both satisfied.

These rules reflect very closely the informal rules we presented previously, modulo the small differences in the formalism mentioned in Section 7.1: We do not need to distinguish between generated/declared patterns and primary patterns in the formalism, as the uniqueness of entities in the primary pattern implies (through name uniqueness, since there is no overloading) the uniqueness of declared entities.

Additionally, DS-Λ makes two overly conservative (and sound) simplifications over our implementation. First, it is possible for the + condition stated by a primary pattern to be mutually exclusive from the nested condition of its own nested pattern. This results in a reflective declaration that can never be expanded, and thus, a range that is disjoint from every other range. We do not check for this in our formalism. Secondly, when two pattern conditions are shown to be mutually exclusive, a one- or two-way unification exists between either the argument types (ME-1), or the argument and return types (ME-2, via SB-R). This unification represents the methods that lie in the intersection of the two ranges defined by the patterns. In practice, we only need to check whether these methods can in fact exist, by checking whether this particular mapping causes any of the remaining patterns to be mutually exclusive. However, in this formalism, we do not apply this unification, and thus check for general mutual exclusion.

7.2.2 *Valid Method Invocation.* The rest of the typing rules add the machinery to standard FGJ type checking to express checks using range containment and disjointness. For instance, to determine valid method invocation is to determine that the reflective environment of the invocation is contained within the reflective environment of the declaration. T-INVK (Figure 10) specifies conditions for a well-typed method invocation. It relies on $\Delta; \Lambda \vdash \mathit{mtype}(\mathtt{n}, \mathtt{T})$ (Figure 13) to handle the complexities in determining the type of method `n` in `T`, where `n` can either be a constant or variable name. We highlight the interesting *mtype* rules.

MT-VAR-R1 and MT-VAR-R2 say that the type of a method with a variable name η in a type `X`, where `X` is either the reflective type for the primary pattern or the reflective type of a *positive* nested pattern, is exactly the type specified by the primary (or nested, respectively) pattern. Otherwise, when looking up a method on a type variable we need to perform the lookup in the type variable's bound (MT-VAR-S).

Reflectively declared methods can also be looked up in a type variable's bound, just as in FGJ (MT-VAR-S). Otherwise, if `T` is a type variable, then we must look for the method type in its bound

Type Unification:

$$\frac{[\overline{\mathtt{U}}/\overline{\mathtt{Z}}]\overline{\mathtt{T}}=[\overline{\mathtt{U}}/\overline{\mathtt{Z}}]\overline{\mathtt{S}} \quad \text{for all } \mathtt{Z}_i\in\overline{\mathtt{Z}},\ \Delta\vdash\mathtt{U}_i\prec:_{\overline{\mathtt{Z}}}\mathtt{Z}_i}{\Delta;[\overline{\mathtt{U}}/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\overline{\mathtt{T}},\ \overline{\mathtt{S}})} \quad \text{(UNI)}$$

Pattern matching rules:

$$\Delta\vdash\mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{T} \quad \text{(PM-REFL)}$$

$$\frac{\Delta\vdash\overline{\mathtt{T}}\prec:_{\overline{\mathtt{Z}}}\overline{\mathtt{S}}}{\Delta\vdash\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\prec:_{\overline{\mathtt{Z}}}\mathtt{C}\texttt{<}\overline{\mathtt{S}}\texttt{>}} \quad \text{(PM-CL)}$$

$$\frac{\mathit{CT}(\mathtt{C})=\mathtt{class\ C}\texttt{<}\overline{\mathtt{X}}\texttt{<}\overline{\mathtt{N}}\texttt{>}\lhd\mathtt{T}\ \{\ldots\} \quad \Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}}{\Delta\vdash\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\prec:_{\overline{\mathtt{Z}}}\mathtt{D}\texttt{<}\overline{\mathtt{S}}\texttt{>}} \quad \text{(PM-CL-S)}$$

$$\frac{\mathtt{Z}\in\overline{\mathtt{Z}} \quad \mathtt{T}\notin\overline{\mathtt{Z}} \quad \mathit{bound}_\Delta(\mathtt{T})=\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \quad \Delta\vdash\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}\prec:_{\overline{\mathtt{Z}}}[\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}/\mathtt{Z}]\mathit{bound}_\Delta(\mathtt{Z})}{\Delta\vdash\mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{Z}} \quad \text{(PM-VAR)}$$

$$\frac{\mathtt{Z}_i\in\overline{\mathtt{Z}} \quad \mathtt{Z}_j\in\overline{\mathtt{Z}} \quad \begin{cases}\Delta\vdash[\mathtt{Z}_i/\mathtt{Z}_j]\mathit{bound}_\Delta(\mathtt{Z}_j)\prec:_{\overline{\mathtt{Z}}}\mathtt{Z}_i & \textit{or}\\ \Delta\vdash[\mathtt{Z}_j/\mathtt{Z}_i]\mathit{bound}_\Delta(\mathtt{Z}_i)\prec:_{\overline{\mathtt{Z}}}\mathtt{Z}_j\end{cases}}{\Delta\vdash\mathtt{Z}_i\prec:_{\overline{\mathtt{Z}}}\mathtt{Z}_j} \quad \text{(PM-PVARS)}$$

Fig. 15. FMJ: Unification and pattern matching functions.

MT-CLASS-R lists conditions for retrieving the type of $\mathtt{n}$ in $\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$, where $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$ has reflectively declared methods. If $\mathtt{n}$ is a name variable, this is simply checking that the range of reference, which is the current reflective environment, is contained within the declaration reflective environment. When $\mathtt{n}$ is a constant name $\mathtt{m}$, however, we need to check whether $\mathtt{m}$ is within the range of method names in the declaration reflective range. We do so by *specializing* the declaration reflective environment using $\mathtt{m}$. Specializing a range based on a constant name is just a way to package the information for uniform use by our containment and disjointness checks (which apply to entire ranges with patterns and not single methods). The main property of $\mathit{specialize}(\mathtt{m},\Lambda_d)=\Lambda_r$ (Figure 12) is that $\mathtt{m}$ is the name of a method in the declaration range, Λ_d, (i.e., a declared method), if and only if the specialized range, Λ_r, is contained by Λ_d. For instance,SP-+ states that, if $\Lambda_d=\langle R_p\,\mathtt{,}\,\mathtt{+}R_n\rangle$, and $\mathtt{m}$ exists in the types reflected over by both R_p and R_n ($\mathtt{T}_i$ and $\mathtt{T}_j$, respectively), then the specialized range has a primary pattern constructed from the actual types of $\mathtt{m}$ in $\mathtt{T}_i$, and a positive nested pattern constructed from the types of $\mathtt{m}$ in $\mathtt{T}_j$. It is then clear from SB-Λ that the specialized range is contained by Λ_d if and only if the types of $\mathtt{m}$ in $\mathtt{T}_i$ and $\mathtt{T}_j$ can be matched by the patterns prescribed by Λ_d—i.e., $\mathtt{m}$ indeed exists in the reflective range Λ_d. If Λ_d has a negative nested pattern, e.g., $\Lambda_d=\langle R_p\,\mathtt{,}\,\mathtt{-}R_n\rangle$, there are two ways for $\mathtt{m}$ to satisfy the negative nested pattern. First, $\mathtt{m}$ can simply be undefined in $\mathtt{T}_j$. Secondly, even if $\mathtt{m}$ is defined in $\mathtt{T}_j$, if its type in $\mathtt{T}_j$ cannot be matched by R_n, it could still be in the range of Λ_d. SP--states that if either of these cases are true, we construct a negative nested pattern by simply applying the unifying substitutions of the primary ranges to $\mathtt{-}R_n$. The resulting specialized range is guaranteed to be contained by Λ_d (by inspection of SB-Λ).

The result of *mtype* is the declared types, with the substitutions of $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$, and the type substitutions for unifying the declaration range and the reference range, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.

Subtyping rules:

$$\Delta \vdash \mathtt{T} <: \mathtt{T} \qquad \text{(S-REFL)}$$

$$\Delta \vdash \mathtt{X} <: \Delta(\mathtt{X}) \qquad \text{(S-VAR)}$$

$$\frac{\Delta \vdash \mathtt{S} <: \mathtt{T} \quad \Delta \vdash \mathtt{T} <: \mathtt{U}}{\Delta \vdash \mathtt{S} <: \mathtt{U}} \qquad \text{(S-TRANS)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C}\langle \overline{\mathtt{X}} \lhd \overline{\mathtt{N}} \rangle \lhd \mathtt{T}\ \{\ldots\}}{\Delta \vdash \mathtt{C}\langle \overline{\mathtt{T}} \rangle <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}} \qquad \text{(S-CLASS)}$$

Fig. 16. FMJ: Subtyping rules.

7.2.3 *Uniqueness of Definitions.* T-METH-R (Figure 10) ensures that methods declared within one reflective block do not conflict with methods in the superclass (i.e., we have proper overriding). The condition is enforced using *override* (Figure 13). *override*($\mathtt{n}$, $\mathtt{T}$, $\overline{\mathtt{U}} \to \mathtt{U}_0$) determines whether method $\mathtt{n}$, defined in some *subclass* of $\mathtt{T}$ with type signature $\overline{\mathtt{U}} \to \mathtt{U}_0$, properly overrides method $\mathtt{n}$ in $\mathtt{T}$. If method $\mathtt{n}$ exists in $\mathtt{T}$, it must have the exact same argument and return types as $\mathtt{n}$ in the subclass. (We made a simplification over FGJ: FGJ allows a covariant return type for overriding methods, whereas we disallow it to simplify the pattern matching rules in Figure 15.) Additionally, the reflective range of $\mathtt{n}$ in the subclass must be either completely contained within one of $\mathtt{T}$'s reflective ranges, or disjoint from all the reflective ranges of $\mathtt{T}$ (and, transitively, $\mathtt{T}$'s superclasses). This condition is enforced using $\Delta \vdash$ *validRange*(Λ, $\mathtt{T}$) (Figure 12).

T-CLASS-R ensures that the reflective blocks within a well-typed class have no declarations that conflict with each other, by requiring ranges of reflective blocks in a class to be disjoint pairwise. Since each block has unique names within itself, the pairwise disjointenss guarantees names across all blocks are unique, as well.

7.3 Soundness

We prove the soundness of FMJ by proving Subject Reduction and Progress for an expression $\mathtt{e}$. Figure 17 defines the operational semantics of FMJ, and Figure 18 defines the method body lookup rules necessary for the operational semantics. Note that method body lookup rules are only defined for lookups under a constant name $\mathtt{m}$. A name variable is only meaningful under a reflective environment Λ. But reduction rules, and thus method body lookup, are done under empty environments, where $\Lambda = \emptyset$. Thus it is not meaningful to define method body lookup for name variable η.

Next, we show the main theorems, important supporting lemmas, and their proof sketches. It should not surprise the reader that the highlighted lemmas are generally those supporting method lookup or invocation, which are the most complex parts of our language and formalism, and also the parts that deviate most from the original FGJ. Detailed proofs can be found in Appendix A.

Theorem 1 [Subject Reduction]. If $\Delta; \Lambda; \Gamma \vdash \mathtt{e} \in \mathtt{T}$, and $\mathtt{e} \to \mathtt{e}'$, then $\Delta; \Lambda; \Gamma \vdash \mathtt{e}' \in \mathtt{S}$ and $\Delta \vdash \mathtt{S} <: \mathtt{T}$ for some $\mathtt{S}$.

Proof Sketch: We prove by structural induction on the reduction rules in

Reduction Rules:

$$\frac{\emptyset \vdash \mathit{fields}(\mathtt{C<\overline{T}>}) = \overline{\mathtt{U}}\ \overline{\mathtt{f}}}{\mathtt{new\ C<\overline{T}>(\overline{e}).f}_i \longrightarrow \mathtt{e}_i} \quad \text{(R-FIELD)}$$

$$\frac{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{new\ C<\overline{T}>(\overline{e}).m(\overline{d}) \longrightarrow [\overline{d}/\overline{x},\ new\ C<\overline{T}>(\overline{e})/this]e_0}} \quad \text{(R-INVK)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{f} \longrightarrow \mathtt{e}_0'.\mathtt{f}} \quad \text{(RC-FIELD)}$$

$$\frac{\mathtt{e}_0 \longrightarrow \mathtt{e}_0'}{\mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \longrightarrow \mathtt{e}_0'.\mathtt{m}(\overline{\mathtt{e}})} \quad \text{(RC-INV-RECV)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i,\ldots) \longrightarrow \mathtt{e}_0.\mathtt{m}(\ldots,\mathtt{e}_i',\ldots)} \quad \text{(RC-INV-ARG)}$$

$$\frac{\mathtt{e}_i \longrightarrow \mathtt{e}_i'}{\mathtt{new\ C<\overline{T}>(\ldots,e}_i\mathtt{,\ldots)} \longrightarrow \mathtt{new\ C<\overline{T}>(\ldots,e}_i'\mathtt{,\ldots)} \quad \text{(RC-NEW-ARG)}$$

Fig. 17. FMJ: Reduction Rules

Method body lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd N\ \{\ldots\ \overline{M}\}} \quad \mathtt{U_0\ m\ (\overline{U}\ \overline{x})\ \{\uparrow e;\} \in \overline{M}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{x}}, \mathtt{e})} \quad \text{(MB-CLASS-S)}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd T\ \{\ldots\ \overline{\mathfrak{M}}\}} \\ \mathfrak{M}_i \in \overline{\mathfrak{M}} \quad \mathfrak{M}_i = \mathtt{<\overline{Y}\lhd\overline{P}>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)\ S_0\ }\eta\mathtt{\ (\overline{S}\ \overline{x})\ \{\uparrow e;\}} \\ \Delta = \overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{N}}, \overline{\mathtt{Y}} \mathtt{<:} \overline{\mathtt{P}} \qquad \Lambda_d = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ \Delta;\emptyset \vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d) = \Lambda_r \qquad \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}] \vdash \Lambda_r \sqsubseteq_\Lambda \Lambda_d \end{array}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{x}}, [\mathtt{m}/\eta]\mathtt{e})} \quad \text{(MB-CLASS-R)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd N\ \{\ldots\ \overline{M}\}} \quad \mathtt{m} \notin \overline{\mathtt{M}}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = \mathit{mbody}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})} \quad \text{(MB-SUPER-S)}$$

$$\frac{\begin{array}{ll} & CT(\mathtt{C}) = \mathtt{class\ C<\overline{X}\lhd\overline{N}>\lhd T\ \{\ldots\ \overline{\mathfrak{M}}\}} \\ \text{for all} & \mathfrak{M}_i \in \overline{\mathfrak{M}}\ \ \mathfrak{M}_i = \mathtt{<\overline{Y}\lhd\overline{P}>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)\ S_0\ }\eta\mathtt{\ (\overline{S}\ \overline{x})\ \{\uparrow e;\}} \\ & \Delta = \overline{\mathtt{X}} \mathtt{<:} \overline{\mathtt{N}}, \overline{\mathtt{Y}} \mathtt{<:} \overline{\mathtt{P}} \qquad \Lambda_d = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathit{reflectiveEnv}(\mathfrak{M}_i)) \\ & \Delta;\emptyset \vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d) = \Lambda_r \\ \text{implies} & \Delta \vdash \mathit{disjoint}(\Lambda_r,\ \Lambda_d) \end{array}}{\mathit{mbody}(\mathtt{m},\ \mathtt{C<\overline{T}>}) = \mathit{mbody}(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})} \quad \text{(MB-SUPER-R)}$$

Fig. 18. FMJ: Method body lookup rules.

Figure 17. The most interesting case is R-INVK, where $\texttt{e}=\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)}$, $\texttt{e}'=\texttt{[}\overline{\texttt{d}}/\overline{\texttt{x}}\texttt{,new C<}\overline{\texttt{T}}\texttt{>/this]e}_0$.

It is easy to see from R-INVK, T-NEW, and T-INVK that,

$mbody(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=(\overline{\texttt{x}},\texttt{e}_0)$ $\quad \Delta\vdash\texttt{new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}\in\texttt{C<}\overline{\texttt{T}}\texttt{>}$ $\quad \Delta\vdash\texttt{C<}\overline{\texttt{T}}\texttt{>}\ ok$

$\Delta;\Lambda\vdash mtype(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=\overline{\texttt{T}}'\to\texttt{T}$ $\quad \Delta;\Gamma;\Lambda\vdash\overline{\texttt{d}}\in\overline{\texttt{S}}$ $\quad \Delta\vdash\overline{\texttt{S}}<:\overline{\texttt{T}}'$

The conclusion follows from the following: 1) the expression $\texttt{e}_0$, obtained via *mbody*, is of type $\texttt{S}'$ where $\Delta\vdash\texttt{S}'<:\texttt{T}$: that is, the body of the method is a subtype of its defined return type (Lemma 1); 2) the expression after term substitution, $\texttt{e}'=\texttt{[}\overline{\texttt{d}}/\overline{\texttt{x}}\texttt{,new C<}\overline{\texttt{T}}\texttt{>/this]e}_0$, is of type S where $\Delta\vdash\texttt{S}<:\texttt{S}'$ (Lemma 4).

Lemma 1: If $\Delta;\Lambda\vdash mtype(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=\overline{\texttt{S}}\to\texttt{S}$, $mbody(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=(\overline{\texttt{x}},\texttt{e}_0)$, and $\Delta\vdash\texttt{C<}\overline{\texttt{T}}\texttt{>}\ ok$, then there exists a type $\texttt{S}'$ such that $\Delta\vdash\texttt{S}'<:\texttt{S}$, $\Delta\vdash\texttt{S}'\ ok$, $\Delta;\Lambda;\overline{\texttt{x}}\mapsto\overline{\texttt{S}},\texttt{this}\mapsto\texttt{C<}\overline{\texttt{T}}\texttt{>}\vdash\texttt{e}_0\in\texttt{S}'$.

Proof Sketch: We prove by induction on the rules of *mbody* (Figure 18). Case MB-CLASS-S does not involve any reflectively declared methods, and thus has a similar proof to that used in FGJ proofs. Cases MB-SUPER-S and MB-SUPER-R can be easily proven through the induction hypothesis. The most interesting case, thus, is MB-CLASS-R, where method body is retrieved from a reflectively declared method $\mathfrak{M}_i$. The corresponding *mtype* is retrieved using rule MT-CLASS-R.

First, notice that there is a bit of ambiguity in MT-CLASS-R (as well as MB-CLASS-R), where $\mathfrak{M}_i$ is defined to be one of $\overline{\mathfrak{M}}$. We first prove using Lemma 2 that there is indeed only one such $\mathfrak{M}_i$ that satisfies the other conditions for the rule to hold.

It is obvious from the correspondence between MT-CLASS-R and MB-CLASS-R, then,

$\Lambda_d=[\overline{\texttt{T}}/\overline{\texttt{X}}](reflectiveEnv(\mathfrak{M}_i))$ $\quad \Delta;\Lambda\vdash specialize(\texttt{m},\ \Lambda_d)=\Lambda_r$ $\quad \Delta;[\overline{\texttt{W}}/\overline{\texttt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$

$\Delta;\Lambda\vdash mtype(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=[\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{W}}/\overline{\texttt{Y}}](\overline{\texttt{S}}''\to\texttt{S}'')$ $\quad mbody(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=[\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{W}}/\overline{\texttt{Y}}](\overline{\texttt{x}}\texttt{,}[\texttt{m}/\eta]\texttt{e})$

Through T-METH-R, we have:

$\Delta''=\overline{\texttt{X}}<:\overline{\texttt{N}},\overline{\texttt{Y}}<:\overline{\texttt{P}}$ $\quad \Lambda''=reflectiveEnv(\mathfrak{M}_i)$ $\quad \Gamma''=\overline{\texttt{x}}\mapsto\overline{\texttt{S}}'',\texttt{this}\mapsto\texttt{C<}\overline{\texttt{X}}\texttt{>}$

$\Delta'';\Lambda'';\Gamma''\vdash\texttt{e}\in\texttt{V}$ $\quad \Delta''\vdash\texttt{V}<:\texttt{S}''$

Thus, to prove that $\Delta;\Lambda;\overline{\texttt{x}}\mapsto\overline{\texttt{S}},\texttt{this}\mapsto\texttt{C<}\overline{\texttt{T}}\texttt{>}\vdash[\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{W}}/\overline{\texttt{Y}}][\texttt{m}/\eta]\texttt{e}\in[\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{W}}/\overline{\texttt{Y}}]\texttt{S}''$, we need to show that 1) type substitutions $[\overline{\texttt{T}}/\overline{\texttt{X}}]$, which are obtained through type-instantiation of a generic class (e.g., instantiating $\texttt{C<}\overline{\texttt{X}}\texttt{>}$ with $\overline{\texttt{T}}$) preserve expression typing as well as subtyping (Lemma 8 and Lemma 6, respectively, which are similarly used in other FGJ-based formalisms), 2) type substitutions $[\overline{\texttt{W}}/\overline{\texttt{Y}}]$ and name substitution $[\texttt{m}/\eta]$, when $[\overline{\texttt{W}}/\overline{\texttt{Y}}]$ and m are results of a *specialize* operation, also preserves typing (Lemma 20).

Lemma 4 (Term Substitution Preserves Typing). If $\Delta;\Lambda;\Gamma,\overline{\texttt{x}}\mapsto\overline{\texttt{T}}\vdash\texttt{e}\in\texttt{T}$, η does not appear in e, $\Delta;\Lambda;\Gamma\vdash\overline{\texttt{d}}\in\overline{\texttt{S}}$, $\Delta\vdash\overline{\texttt{S}}<:\overline{\texttt{T}}$, then $\Delta;\Lambda;\Gamma\vdash[\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e}\in\texttt{T}'$, for some $\texttt{T}'$ where $\Delta;\Lambda;\Gamma\vdash\texttt{T}'<:\texttt{T}$.

Proof Sketch: We prove by induction on expression typing rules T-* (Figure 10). The most interesting case is T-INVK, where $\texttt{e}=\texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)}$. By induction hypothesis,

$\Delta;\Lambda;\Gamma\vdash\texttt{e}_0\in\texttt{T}_0$ $\quad \Delta;\Lambda;\Gamma\vdash[\overline{\texttt{d}}/\overline{\texttt{x}}]\texttt{e}_0\in\texttt{T}'_0$ $\quad \Delta\vdash\texttt{T}'_0<:\texttt{T}_0$

We need to show that $\Delta;\Lambda\vdash mtype(\texttt{m},\ \texttt{T}_0)=\overline{\texttt{T}}\to\texttt{T}$ implies $\Delta;\Lambda\vdash mtype(\texttt{m},\ \texttt{T}'_0)=\overline{\texttt{T}}\to\texttt{T}$.

That is to say, the method `m` is defined for a superclass, then it is defined in the subclass, as well, with the same type signature. Lemma 19, which is a standard lemma for FGJ-based formalisms, proves this point. However, in our proofs of Lemma 19, we needed to prove two additional, interesting lemmas regarding the properties of range containment. Lemma 16 shows that type substitutions preserve single range containment; Lemma 23 shows that single range containment is transitive. The preservation of reflective range (i.e., primary and nested) containment and transitivity then follows easily from these lemmas regarding single ranges.

Lemma 16 (Substitution Preserves Single Range Containment): If $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2$, $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, where $\Delta_1\vdash\overline{\mathtt{U}}$ ok, and none of $\overline{\mathtt{X}}$ appears in Δ_1, none of $\overline{\mathtt{X}}$ appears on $\overline{\mathtt{Y}}$, then $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2$, where $\overline{\mathtt{W}}'=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.
Lemma 23 (Single Range Containment is Transitive): If $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1 \sqsubseteq_R R_2$, and $\Delta;[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_2 \sqsubseteq_R R_3$, then $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_1 \sqsubseteq_R R_3$,
Proof Sketch: Both proofs follow from analysis of rule SB-R.

Theorem 2 [Progress]. Let `e` be a well-typed expression. 1. If `e` has `new C<`$\overline{\mathtt{T}}$`>(`$\overline{\mathtt{e}}$`).f` as a subexpression, then $\emptyset\vdash$*fields*(`C<`$\overline{\mathtt{T}}$`>`)=$\overline{\mathtt{U}}\ \overline{\mathtt{f}}$, and `f` = $\mathtt{f}_i$. 2. If `e` has `new C<`$\overline{\mathtt{T}}$`>(`$\overline{\mathtt{e}}$`).m(`$\overline{\mathtt{d}}$`)` as a subexpression, then *mbody*(`m`, `C<`$\overline{\mathtt{T}}$`>`)=($\overline{\mathtt{x}}$,$\mathtt{e}_0$) and $|\overline{\mathtt{x}}| = |\overline{\mathtt{d}}|$.
Proof Sketch: Proof follows from T-FIELD and T-INVK respectively, and from the well-typedness of subexpressions.

Theorem 3 [Type Soundness]. If $\emptyset;\emptyset;\emptyset\vdash\mathtt{e}\in\mathtt{T}$ and $\mathtt{e}\longrightarrow^*\mathtt{e}'$, then $\mathtt{e}'$ is a value `v` such that $\emptyset;\emptyset;\emptyset\vdash\mathtt{v}\in\mathtt{S}$ and $\emptyset\vdash\mathtt{S}<:\mathtt{T}$ for some type `S`.
Proof: Follows from Theorems 1 and 2.

8. DISCUSSION

We next discuss briefly some interesting issues concerning MorphJ and its type checking.

8.1 Expressiveness, Completeness, and Decidability

MorphJ is a limited reflection language and certainly does not compare in expressiveness to powerful pattern matching and transformation languages (e.g., [Mens et al. 2001; Visser 2004]). The goal of MorphJ is to facilitate a specific, important kind of transformation with desirable properties, such as modular type-checking. Furthermore, our type system is not complete: there are cases of range disjointness or containment that are true (due to some logical combination of properties of Java typing and the types at hand) yet our approach cannot prove. This is true of both our formalism and our MorphJ implementation. (For instance, we have no rule to infer that a positive nested pattern implies a negative one, although this can be the case since our formalism allows no overloading.) Such corner cases are obscure, however—our unification-based checks cover the generally interesting cases of containment and disjointness we could identify. Even with its focused scope, however, the MorphJ type system flirts with undecidability, since it adds explicit iteration at compile-time. Many of our restrictions (such as disallowing nesting of primary patterns and preventing type variables bound in a filter pattern from appearing in generated code) are aimed precisely at keeping the type-checking manageable.

Still, subtyping and type recursion introduce potential undecidability. There could be circularity in mixin definitions, in method definitions, in reflection sets, etc. For instance, consider:

```
class C<X extends D<X>> {
  <R>[m]for(R m() : X.methods) {|...|}
}
class D<X> extends C<D<X>> { ... }
```

The methods of `C<X>` are circularly defined: they reflect over the methods of `X`, which include the methods of `D<X>`, which, in turn, include the methods of `C<D<X>>`. Similarly, we could have (in the full MorphJ, although not in the formalism, since the reflection set cannot be complex):

```
class C<X> {
  <R>[m]for(R m() : D<X>.methods) {|...|}
}
class D<X> {
  <R>[m]for(R m() : C<C<X>>.methods) {|...|}
}
```

In the above example, `C<X>`'s methods are defined by reflecting over `D<X>`'s methods, which in turn depend on `C<C<X>>`'s methods, and so on. We do not expect that MorphJ's expressiveness will depend crucially on type-recursion, so we just reject cyclic type references conservatively. That is, our type checker keeps track of classes with reflectively declared members, and maintains a stack of them during every type reasoning action. If the same such class is encountered twice (even with different arguments) the code entity being checked is conservatively rejected.

8.2 MorphJ Implementation

It should not be a surprise to readers that MorphJ cannot be implemented using an erasure-based approach, as is adopted in the implementation of Java generics. In an erasure-based implementation, all type-instantiations of a generic share the same copy of bytecode. The type variables of a generic are "erased" to their upper bounds in that shared bytecode. Appropriate casts are inserted at the use-sites of type-instantiations. This particular approach works for Java generics because the *structure* of a Java generic does not vary with its type-instantiations—a generic always declares the same methods, fields, and superclasses regardless of its type arguments. A MorphJ generic class, however, may declare different methods and fields, as well as different supertypes, depending on its type arguments. Thus, it is impossible for different type-instantiations of a MorphJ generic to share the same copy of bytecode. MorphJ is thus implemented via expansion: Every type-instantiation of a MorphJ generic is expanded to its own bytecode representation.

Unlike the expansion-based approach taken by C++ templates, a MorphJ generic is still separately compiled. After type-checking, a MorphJ generic is compiled to an annotated bytecode file. The bytecode sequences corresponding to reflectively declared code are annotated with information pertaining to the associated reflective iterator: patterns, nested patterns, and pattern-matching variables. This copy of bytecode is then used as a template for expansion. Upon type-instantiation, the annotated bytecode sequences are expanded using information in the annotations.

For instance, a block of bytecode for a reflectively declared method may be expanded to bytecode for multiple methods, by replacing type and name variables in the original with concrete types and identifiers. Alternatively, a block of bytecode may be removed entirely if the range of its associated reflective iterator is empty. Type-checking of the expanded bytecode is not necessary, since MorphJ's source-level type-checking guarantees that any type-instantiation of a generic is always well-typed. The bytecode of a MorphJ generic class, assuming it has reflective declarations, is not considered valid by the Java bytecode verifier. Its expanded versions, however, are.

MorphJ is implemented using the JastAdd extensible compiler framework for Java [Ekman and Hedin 2007], and ASM Java bytecode library [Bruenton et al.]. MorphJ is available via *http://code.google.com/p/morphing/*.

9. RELATED WORK

The limitations stemming from rigidity in code structure have been the focus of much previous research, starting as early as the design of Lisp macros in the 1960's. Morphing can be seen as the latest step in this line of research. We next discuss morphing and MorphJ relative to past work.

9.1 Static Type Conditions vs. Morphing

Our recent work on statically safe type conditions [Huang et al. 2007a] has enough relation to morphing features to warrant a detailed comparison. The cJ language is an extension of Java with a static-if construct, allowing the configuration of generic classes based on properties of their type parameters. For instance, cJ can express a `List<X>` class that implements `Serializable` only when its type parameter `X` implements `Serializable`. MorphJ adds a similar reflective "if". In addition, MorphJ adds a reflective "for", as well as the ability to create declarations with non-constant names. Thus, MorphJ is a more ambitious language with significantly more complexity, which is reflected in the different design focus and implementation decisions for cJ and MorphJ.

The cJ reflective "if" is based on subtyping conditions in a nominal subtyping system, whereas the MorphJ reflective "if" is based on structural constraints, e.g., whether a type has a method or field matching a certain pattern. But more importantly, the cJ "if" conditions are restricted to straightforward subtyping and conjunctions of multiple subtyping conditions. MorphJ, however, allows the expression of negative conditions, using "`if ( no` ***`PATTERN`*** `)`" clauses. MorphJ uses negative conditions extensively to ensure the absence of conflicting declarations.

cJ is designed with backward compatibility in mind, enabling an erasure-based translation. cJ language constructs can be "erased", producing regular Java code in a one-to-one correspondence between cJ generic classes and Java generic classes. Additionally, cJ interacts smoothly with advanced features in the Java type system, such as variance [Torgersen et al. 2004; Igarashi and Viroli 2006] and polymorphic methods. In contrast, MorphJ takes a more radical approach, favoring feature-richness and integration of ideas over backward compatibility and implementation integration. This difference is most evident in MorphJ's implementation, which employs an expansion-based translation.

9.2 Comparison to Traditional Meta-programming Techniques

Meta-programming techniques offer the ability for programs to generate other programs, allowing the structure of generated programs to change based an arbitrary conditions. Instances of such techniques are macro facilities such as Lisp macros, reflection, meta-object protocols [Danforth and Forman 1994; Kiczales et al. 1991], or pattern-based program generation and transformation [Bachrach and Playford 2001; Baker and Hsieh 2002; Visser 2004; Batory et al. 1998]. The goal of structural abstraction is to promote meta-programming capabilities offered in these low-level mechanisms to high-level language features, with support for full modular type-safety. None of the above mechanisms offer such safety guarantees: A macro or meta-class cannot be type-checked independently from their compositions, in a way that guarantees it is well-typed for all its possible compositions.

An interesting special case of program generation is *staging languages* such as MetaML [Taha and Sheard 1997] and MetaOCaml [Calcagno et al. 2003]. These languages offer modular type safety: The generated code is guaranteed correct for any input, if the generator type-checks. Nevertheless, MetaML and MetaOCaml do not allow generating identifiers (e.g., names of variables) or types that are not constant (as is supported in MorphJ). Generally, staging languages target program specialization rather than full program generation: The program must remain valid even when staging annotations are removed. Thus, staging programs do not abstract over the structure of programs—they are quite explicit in the structure of the generated program, in fact. It is interesting that even recent meta-programming tools, such as Template Haskell [Sheard and Jones 2002] are explicitly not modularly type safe—its authors acknowledge that they sacrifice the MetaML guarantees for expressiveness.

9.3 Comparison to Efforts in Safe Program Generation/Transformation

Recent mechanisms such as Genoupe [Draheim et al. 2005], SafeGen [Huang et al. 2005], and compile-time reflection (CTR) [Fähndrich et al. 2006] attempt to add safety guarantees to meta-programming, while maintaining expressiveness. Nevertheless, these approaches either fail to achieve full safety, or reject programs in a way that is not transparent to the programmer. For instance, the Genoupe approach has been shown unsafe, as the reasoning depends on properties that can change at runtime; SafeGen has no soundness proof and relies on the capabilities of an automatic theorem prover—an unpredictable and unfriendly process from the programmer's perspective.

MorphJ's closest relative is CTR [Fähndrich et al. 2006]. CTR is an extension to C# that pioneered the use of patterns for reflective iteration and was one of the first systems to aim for modular type safety. Nevertheless, its modular guarantees concern only validity of references and not the absence of declaration conflicts. A unique aspect of CTR (compared to MorphJ) is that it transforms classes in-place, which enables some interesting applications. MorphJ, on the other hand, only allows enhancement of classes through subtyping or delegation. MorphJ improves over CTR, however, by adding more expressiveness through nested patterns, while keeping or strengthening the typing guarantees. For instance, CTR does not allow matching multiple method argument types, or existential conditions on iteration

ranges.

In addition to lacking full type safety, neither of these mechanisms integrate seamlessly with a programming language, as MorphJ does. All these approaches require an concept outside the base language, such as a "generator" or a "transform". In MorphJ, the concept of code generation is incorporated with the concept of generic classes. Additionally, these mechanisms use complex syntax for retrieving reflective members, whereas MorphJ utilizes patterns very similar to method and field signatures.

9.4 Comparison to AOP Tools

Morphing can be seen as an alternative for a central part of AOP functionality: aspect advice of structural program features, such as method before-, after-, and around-advice. Particularly, the logging example in Section 1 and the synchronization example in Section 3 are frequent use cases for AOP languages. Compared to major AOP languages, such as AspectJ [Kiczales et al. 2001], morphing has some significant differences. These include the way functionality is added (AOP mechanisms modify the original class and have no concept of unmodified and modified class as separate types in the same program); the way the matching is performed (e.g., MorphJ allows matching using subtype-based semantic conditions, in contrast to AspectJ's purely syntactic matching); and the way structural reflection is controlled (morphing requires explicit parameterization so, for instance, order-of-application issues are left to the programmer). The main difference, however, is MorphJ's guarantee of modular type safety. Generally, morphing tries to offer structural reflection, in a way that is closely tied to the static type system, and allows modular reasoning without reference to the morphed class's instantiations.

9.5 Other Static Reflection

An extension of traits [Reppy and Turon 2007] offers pattern-based reflection by allowing a trait to use name variables for declarations. However, [Reppy and Turon 2007] does not offer static iteration over the members of classes—a name-generic trait must be mixed in once for each name instance.

There has been a line of work focused on providing statically type-safe generic traversal of data structures [Lämmel and Jones 2003; Jansson and Jeuring 1997; Gibbons 2007]. For instance, the "scrap your boilerplate" [Lämmel and Jones 2003] line of work offers extensions of Haskell that allow code to abstract over the exact structure of the data types it acts on, and to have the appropriate functions invoked when their expected data types are encountered during traversals. Abstracting over the structures of data types in functional languages is similar to abstracting over the fields and methods of classes in object-oriented languages. [Weirich and Huang 2004] offers such generic traversal capabilities for Java. However, whereas [Lämmel and Jones 2003; Jansson and Jeuring 1997; Weirich and Huang 2004] focus on offering structurally-generic *traversal*, MorphJ focuses on structurally-generic *declarations*. Neither of [Lämmel and Jones 2003; Jansson and Jeuring 1997; Weirich and Huang 2004] allow more functions to be declared using the names or types retrieved from a non-specific data type. Thus, these techniques fall short of MorphJ (and static reflection work in general [Draheim et al. 2005; Fähndrich et al. 2006; Huang et al. 2005]) in this respect. On the other hand, MorphJ is

not well-suited for writing generic traversal code. Traversing data structures and invoking methods on objects encountered is largely based on the dynamic types of these objects. MorphJ's reflective declarations are based purely on the static types of fields and methods.

Another related line of work in the functional programming language community is type-indexed types [Hinze et al. 2004; Chakravarty et al. 2005; Chakravarty et al. 2005; Schrijvers et al. 2008]. Type-indexed types are generic data types whose structure can be specialized by inducting over the structure of their type parameters. For instance, a data type `Map<T>` can be defined to be some existing type `MapChar` if `T` is instantiated with a base type `Char`. If `T` is instantiated with a data type whose structure comprises the base type `Char`, and some other type `S`, (e.g., the structure of `String` can comprise a `Char` and another `String`) then `Map<T>` can be defined by composing the resulting types of `Map<Char>` and `Map<S>`. How types are composed can be defined generically for all possible types `T`, by pattern-matching on the structure of `T`. Type-indexed types, however, do not allow the manipulation of *names* associated with the structural components of type parameters. For instance, if there are labels associated with components of `String`: {*`firstChar`*`: Char, `*`rest`*`: String`}, one cannot define a type-indexed type that systematically uses those same labels in the composed type. Manipulation of names and guaranteeing type safety in the presence of name manipulation, on the other hand, is one of the defining features of morphing.

10. DISCUSSION AND CONCLUSIONS

Morphing can form a sound foundation for a new paradigm of programming, where code entities are no longer rigid and fixed but can shape their structure by inspecting other code entities. This opens the door for rethinking many of the existing mechanisms for code reuse and language interaction. A clear instance concerns the possible impact of morphing on the fundamental "inheritance vs. delegation" dilemma. We offer some ideas next, and intend to pursue this direction further in future work.

The dominant mechanism for code reuse in most object-oriented languages is inheritance: class `A extends B`, and thus `A` inherits (and reuses) all the code in `B`. The criticisms of inheritance as a mechanism for reuse are well-documented in the research literature [Cook et al. 1990; Ducasse et al. 2006]. Many OO languages confuse inheritance and subtyping, also confusing the role of a class as a model for object behavior and its role as an organizational unit of code. When `A` inherits code from `B`, `A` is automatically treated as a subtype, or a refinement of the model, of `B`. This may not be the desirable behavior—`A` may simply want to reuse some code. Inheritance is also a coarse-grained way for code reuse: Oftentimes a subclass only needs to reuse a small subset of its superclass's methods, but is forced to inherit all of them. Various alternative reuse mechanisms have been proposed, such as mixins, traits, etc. [Ducasse et al. 2006; Cannon 1982; Bracha and Cook 1990]. These techniques either have trouble providing modular safety guarantees [Cannon 1982; Bracha and Cook 1990], or they have limitations with respect to encapsulating and sharing state [Ducasse et al. 2006]. (An overall problem is that of evolution: how do changes to a given class or mixin impact other parts of the class hierarchy?

Modular type safety can be seen as a projection of this problem.)

An alternative model of code reuse is delegation [Lieberman 1986; Stein 1987]. Instead of inheriting code from class `B`, class `A` defines methods that model its own behavior, and delegates method calls to the appropriate methods in `B` if the behavior is shared. In the delegation approach, `A` is not treated as a subtype of `B`. `A` can also reuse as much or as little of `B`'s code as it sees fit. The synchronization proxies in Java Collections Framework described in Section 3 can be seen as code reuse through delegation. However, as is also evident through the synchronization proxies, delegation involves an enormous amount of boiler plate delegation method declarations. Furthermore, if `A` needs to reuse code from multiple classes through delegation, keeping track of state shared between different delegates is cumbersome and error-prone. Additionally, there is some performance penalty involved in each delegation.

Morphing presents a fresh perspective to the inheritance vs. delegation debate. Using morphing, class `A` can easily share code with class `B` by defining its methods (or fields) through reflective iteration over methods (or fields) of `B`. The body of these methods may delegate calls to the methods of `B`, which emulates the classic delegation approach. Code reuse does not confuse itself with the concept of modeling: Class `A` does not have to be a subtype of `B` to reuse code from `B`. Yet boiler-plate code is replaced with one reflective iteration block.

Alternatively, syntax could be introduced to indicate to the compiler that the code from `B` should actually be copied into the body of `A`, which more closely emulates inheritance. State variables of `B` can be explicitly kept inside of `A`. The overhead of delegation calls can also be removed, since the code of `B` is now inlined in `A`, instead of being delegated to.

Furthermore, morphing allows code reuse at a granularity that is defined by the programmer. Class `A` can choose to inherit all the methods in `B`, or only a certain subset of methods in `B` by refining the patterns used in reflective iteration over `B`. `A` may also choose to inherit methods of `B`, as well as methods from `C`, `D`, etc., using multiple reflective iteration blocks. This provides an emulation of "multiple inheritance". But, in the morphing approach, programmers have explicit control over how methods from multiple classes are incorporated and interact. For instance, programmers can explicitly state that methods with the same signature from `B`, `C`, and `D` should first invoke the code of `B`, then `D`, then `C` (or any arbitrary order, or even leave out invocation of code of `C`). Similarly, state variables from multiple classes can be kept separate, or combined if they have the same name and type. In fact, this new form of inheritance can be generally defined as a MorphJ generic class `A<X>`, where `A` can inherit from any type `X`. We can similarly define generic classes `A<X,Y>`, or `A<X,Y,Z>`, etc. for generic "multiple inheritance".

Thus, morphing is capable of supporting a hybrid approach between inheritance and delegation, where code reuse is kept a separate concept from modeling, granularity of code reuse can be explicitly controlled by the programmers, semantics of shared code and state can be explicitly managed, and delegation costs can be optimized away.

To conclude, morphing represents a significant trend in the evolution of programming languages. Most major advances in programming languages are modularity or

reusability enhancements. The first step was taken with *procedural abstraction* in the 50s and 60s, which culminated in structured programming languages. Procedural abstraction captured algorithmic logic in a form that could be multiply reused both in the same program and across programs, over different data objects. The next major abstraction step was arguably *type abstraction* or *polymorphism*, which allowed the same abstract logic to be applied to multiple types of data, although the low-level code for each type would end up being substantially different. Morphing may very well be the next big step in language evolution, where code can abstract over the structure of other program elements. We expect that the inclusion of such constructs in mainstream languages will be a topic of major importance for decades to come.

Acknowledgments. This work was funded by the NSF (CCF-0917774, CCF-0934631) and by LogicBlox Inc. We would like to thank the TOPLAS reviewers, as well as reviewers for ECOOP'07 and PLDI'08 and David Zook for their thoughtful comments and suggestions.

REFERENCES

Allen, E., Bannet, J., and Cartwright, R. 2003. A first-class approach to genericity. In *OOPSLA 2003: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York, NY, 96–114.

Apache Software Foundation. *Byte-code engineering library. "http://jakarta.apache.org/bcel /manual.html"*. Accessed June 2009.

Bachrach, J. and Playford, K. 2001. The Java syntactic extender (JSE). In *OOPSLA 2001: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York, NY, 31–42.

Baker, J. and Hsieh, W. C. 2002. Maya: multiple-dispatch syntax extension in Java. In *PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*. ACM Press, New York, NY, 270–281.

Batory, D., Lofaso, B., and Smaragdakis, Y. 1998. JTS: tools for implementing domain-specific languages. In *Proceedings of the Fifth International Conference on Software Reuse*. IEEE, New York, NY, 143–153.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. In *OOPSLA/ECOOP 1990: Proceedings of the European Conference on Object-Oriented Programming and Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York, NY, 303–311.

Bruenton, E. et al. *ASM Java Bytecode Engineering Library: http://asm.ow2.org/*. Accessed June 2009.

Calcagno, C., Taha, W., Huang, L., and Leroy, X. 2003. Implementing multi-stage languages using ASTs, gensym, and reflection. In *GPCE 2003: Proceedings the 2nd International Conference on Generative Programming and Component Engineering*. Springer-Verlag New York, Inc., New York, NY, 57–76.

Cannon, H. I. 1982. Flavors: A non-hierarchical approach to object-oriented programming. Tech. rep.

Chakravarty, M. M. T., Keller, G., and Jones, S. P. 2005. Associated type synonyms. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 241–253.

Chakravarty, M. M. T., Keller, G., Jones, S. P., and Marlow, S. 2005. Associated types with class. In *In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1–13.

Cook, W. R., Hill, W., and Canning, P. S. 1990. Inheritance is not subtyping. In *POPL 1990: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 125–135.

Csallner, C. and Smaragdakis, Y. 2004. JCrasher: An automatic robustness tester for Java. *Software—Practice and Experience 34,* 11 (Sept.), 1025–1050.

Danforth, S. and Forman, I. R. 1994. Reflections on metaclass programming in SOM. In *OOPSLA 1994: Proceedings of the 9th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, New York, NY, 440–452.

Draheim, D., Lutteroth, C., and Weber, G. 2005. A type system for reflective program generators. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. LNCS 3676. Springer-Verlag, Heidelberg, Germany, 327–341.

Ducasse, S., Nierstrasz, O., Schärli, N., Wuyts, R., and Black, A. P. 2006. Traits: A mechanism for fine-grained reuse. *TOPLAS: ACM Transactions on Programming Languages and Systems 28,* 2, 331–388.

Ekman, T. and Hedin, G. 2007. The JastAdd extensible Java compiler. In *OOPSLA 2007: Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, 1–18.

Fähndrich, M., Carbin, M., and Larus, J. R. 2006. Reflective program generation with patterns. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. ACM Press, New York, NY, 275–284.

Gamma, E., Helm, R., and Johnson, R. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley.

Gibbons, J. 2007. In *Spring School on Datatype-Generic Programming*, R. Backhouse, J. Gibbons, R. Hinze, and J. Jeuring, Eds. LNCS, vol. 4719. Springer-Verlag.

Herlihy, M., Luchangco, V., and Moir, M. 2006. A flexible framework for implementing software transactional memory. In *OOPSLA 2006: Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, 253–262.

Hinze, R., Jeuring, J., and Löh, A. 2004. Type-indexed data types. *Sci. Comput. Program. 51,* 1-2, 117–151.

Huang, S. S. and Smaragdakis, Y. 2006. Easy language extension with Meta-AspectJ. In *ICSE 2006: Proceedings of International Conference on Software Engineering*. ACM, New York, NY, 865–868.

Huang, S. S. and Smaragdakis, Y. 2008. Expressive and safe static reflection with MorphJ. In *PLDI 2008: Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*. Vol. 43. ACM, New York, NY, 79–89.

Huang, S. S., Zook, D., and Smaragdakis, Y. 2005. Statically safe program generation with SafeGen. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering*. LNCS 3676. Springer-Verlag, 309–326.

Huang, S. S., Zook, D., and Smaragdakis, Y. 2007a. cJ: Enhancing Java with safe type conditions. In *Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development*. ACM Press, Vancouver, British Columbia, Canada, 185–198.

Huang, S. S., Zook, D., and Smaragdakis, Y. 2007b. Morphing: Safely shaping a class in the image of others. In *ECOOP 2007: 21st European Conference on Object Oriented Programming*, E. Ernst, Ed. Lecture Notes in Computer Science, vol. 4609. Springer, 303–329.

Igarashi, A., Pierce, B., and Wadler, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *TOPLAS: ACM Transactions on Programming Languages and Systems 23,* 3, 396–450.

Igarashi, A. and Viroli, M. 2006. Variant parametric types: A flexible subtyping scheme for generics. *TOPLAS: ACM Transactions on Programming Languages and Systems 28,* 5, 795–847.

Jansson, P. and Jeuring, J. 1997. PolyP - a polytypic programming language extension. In *POPL 1997: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, New York, NY, 470–482.

Kiczales, G., des Rivieres, J., and Bobrow, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. 2001. An overview of AspectJ. In *ECOOP 2001: Proceedings of the 15th European Conference on Object Oriented Programming*. Springer-Verlag, London, UK, 327–353.

Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. 1997. Aspect-Oriented Programming. In *ECOOP 1997: Proceedings of the 11th European Conference on Object Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Vol. 1241. Springer-Verlag, Heidelberg, Germany, and New York, 220–242.

Lämmel, R. and Jones, S. P. 2003. Scrap your boilerplate: a practical design pattern for generic programming. In *TLDI 2003: Proceedings of the 2003 ACM SIGPLAN International workshop on Types in Languages Design and Implementation*. ACM, New York, NY, 26–37.

Lieberman, H. 1986. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not. 21,* 11, 214–223.

Mens, K., Michiels, I., and Wuyts, R. 2001. Supporting software development through declaratively codified programming patterns. In *13th Int'l Conf. Software Engineering and Knowledge Engineering, Buenos Aires*. Knowledge Systems Institute, 136–143.

Mohnen, M. 2002. Interfaces with default implementations in Java. In *Proceedings of the Inaugural Conference on the Principles and Practice of Programming*. National University of Ireland, Maynooth, County Kildare, Ireland, Ireland, 35–40.

Reppy, J. and Turon, A. 2007. Metaprogramming with traits. In *ECOOP 2007: Proceedings of the European Conference on Object Oriented Programming*. Springer-Verlag, Berlin,Germany, 373–398.

Schrijvers, T., Peyton Jones, S., Chakravarty, M., and Sulzmann, M. 2008. Type checking with open type functions. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN international conference on Functional programming*. ACM, New York, NY, USA, 51–62.

Sheard, T. and Jones, S. P. 2002. Template meta-programming for Haskell. In *Proceedings of the ACM SIGPLAN Workshop on Haskell*. ACM Press, Pittsburgh, Pennsylvania, 1–16.

Smaragdakis, Y. and Batory, D. 1998. Implementing layered designs with mixin layers. In *ECOOP 1998: Proceedings of the 12th European Conference on Object Oriented Programming*. Springer-Verlag LNCS 1445, 550–570.

Stein, L. A. 1987. Delegation is inheritance. In *OOPSLA 1987: Proceedings on ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, 138–146.

Taha, W. and Sheard, T. 1997. Multi-stage programming with explicit annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and semantics-based Program Manipulation*. ACM Press, Amsterdam, The Netherlands, 203–217.

Torgersen, M., Hansen, C. P., Ernst, E., von der Ahe, P., Bracha, G., and Gafter, N. 2004. Adding wildcards to the java programming language. In *SAC 2004: Proceedings of the 2004 ACM Symposium on Applied Computing*. ACM Press, Nicosia, Cyprus, 1289–1296.

Visser, E. 2004. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In *Domain-Specific Program Generation*, C. Lengauer, D. Batory, C. Consel, and M. Odersky, Eds. Springer-Verlag, 216–238. LNCS 3016.

Weirich, S. and Huang, L. 2004. A design for type-directed Java. In *Workshop on Object-Oriented Developments (WOOD)*, V. Bono, Ed. ENTCS.

APPENDIX

A. FEATHERWEIGHT MORPHJ (FMJ): PROOF OF SOUNDNESS

Theorem 1 Subject Reduction. *If* $\Delta;\Lambda;\Gamma \vdash \mathtt{e}\in\mathtt{T}$ *and* $\mathtt{e} \rightarrow \mathtt{e}'$*, then for some* $\mathtt{S}$*,* $\Delta;\Lambda;\Gamma \vdash \mathtt{e}'\in\mathtt{S}$ *and* $\Delta \vdash \mathtt{S}<:\mathtt{T}$.

Proof. Prove by structural induction on the reduction rules.

Case R-FIELD: $\mathtt{e}=\mathtt{new\ C}\langle\overline{\mathtt{T}}\rangle(\overline{\mathtt{e}})\mathtt{.f}_i$, $\mathtt{e}'=\mathtt{e}_i$
By T-NEW,

$\Delta;\Lambda;\Gamma\vdash$`new C<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`)`$\in$`C<`$\overline{\texttt{T}}$`>` $\quad \Delta\vdash \mathit{fields}($`C<`$\overline{\texttt{T}}$`>`$)=\overline{\texttt{U}}\ \overline{\texttt{f}}$
$\Delta;\Lambda;\Gamma\vdash \overline{\texttt{e}}\in\overline{\texttt{S}}$ $\quad \Delta\vdash\overline{\texttt{S}}<:\overline{\texttt{U}}$

By T-FIELD and definition of *bound*,

$\mathit{bound}_\Delta($`C<`$\overline{\texttt{T}}$`>`$)=$`C<`$\overline{\texttt{T}}$`>` $\quad \Delta;\Lambda;\Gamma\vdash$`new C<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`).f`$_i\in$`U`$_i$

Let `S` be `S`$_i$, `T` be `U`$_i$.

Case R-INVK: `e=new C<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`).m(`$\overline{\texttt{d}}$`)`, `e`$'$`=[`$\overline{\texttt{d}}/\overline{\texttt{x}}$`,new C<`$\overline{\texttt{T}}$`>/this]e`$_0$

By R-INVK, T-NEW, T-INVK

$\mathit{mbody}($`m, C<`$\overline{\texttt{T}}$`>`$)=(\overline{\texttt{x}},\texttt{e}_0)$ $\quad \Delta\vdash$`new C<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`)`$\in$`C<`$\overline{\texttt{T}}$`>` $\quad \Delta\vdash$`C<`$\overline{\texttt{T}}$`>` *ok*
$\Delta;\Lambda\vdash \mathit{mtype}($`m, C<`$\overline{\texttt{T}}$`>`$)=\overline{\texttt{T}}'\to$`T` $\quad \Delta;\Gamma;\Lambda\vdash\overline{\texttt{d}}\in\overline{\texttt{S}}$ $\quad \Delta\vdash\overline{\texttt{S}}<:\overline{\texttt{T}}'$

By Lemma 1, for some `S`, $\Delta\vdash$`S`$<:$`T`, $\Delta\vdash$`S` *ok*, $\Delta;\Lambda;\overline{\texttt{x}}\mapsto\overline{\texttt{T}}'$,`this`$\mapsto$`C<`$\overline{\texttt{T}}$`>`$\vdash$`e`$_0\in$`S`.

By Lemma 4, for some `S`$'$ where $\Delta;\Lambda;\Gamma\vdash$`S`$'<:$`S` ,

$\Delta;\Lambda;\Gamma\vdash[\overline{\texttt{d}}/\overline{\texttt{x}}$`,new C<`$\overline{\texttt{T}}$`>/this]e`$_0\in$`S`$'$,

Case RC-FIELD: `e=e`$_0$`.f`, `e`$'$`=e`$'_0$`.f`

By T-FIELD,

$\Delta;\Lambda;\Gamma\vdash$`e`$_0\in$`T`$_0$ $\quad \Delta\vdash \mathit{fields}(\mathit{bound}_\Delta($`T`$_0$`))=\overline{\texttt{T}}\ \overline{\texttt{f}}$ $\quad \Delta;\Lambda;\Gamma\vdash$`e`$_0$`.f`$\in$`T`$_i$

By induction hypothesis,

$\Delta;\Lambda;\Gamma\vdash$`e`$'_0\in$`S`$_0$ $\quad \Delta\vdash$`S`$_0<:$`T`$_0$

By Lemma 11, $\Delta\vdash \mathit{fields}(\mathit{bound}_\Delta($`S`$_0$`))=\overline{\texttt{S}}\ \overline{\texttt{g}}$, $\Delta;\Lambda;\Gamma\vdash$`e`$'_0$`.f`$\in$`S`$_i$, `S`$_i$`=T`$_i$.

By S-REFL, $\Delta\vdash$`S`$_i<:$`T`$_i$.

Let `T` be `T`$_i$, `S` be `S`$_i$.

Case RC-INV-RECV: `e=e`$_0$`.m(`$\overline{\texttt{e}}$`)`, `e`$'$`=e`$'_0$`.m(`$\overline{\texttt{e}}$`)`

By T-INVK,

$\Delta;\Lambda;\Gamma\vdash$`e`$_0\in$`T`$_0$ $\quad \Delta;\Lambda;\Gamma\vdash\overline{\texttt{e}}\in\overline{\texttt{T}}'$
$\Delta;\Lambda\vdash \mathit{mtype}($`m, T`$_0$`)`$=\overline{\texttt{T}}\to$`T` $\quad \Delta\vdash\overline{\texttt{T}}'<:\overline{\texttt{T}}$

By induction hypothesis, Lemma 19, and T-INVK,

$\Delta;\Lambda;\Gamma\vdash$`e`$'\in$`S`$_0$ $\quad \Delta\vdash$`S`$_0<:$`T`$_0$
$\Delta;\Lambda\vdash \mathit{mtype}($`m, S`$_0$`)`$=\overline{\texttt{T}}\to$`T` $\quad \Delta;\Lambda;\Gamma\vdash$`e`$'$`.m(`$\overline{\texttt{e}}$`)`$\in$`T`

Let `S` be `T`.

Case RC-INV-ARG: Easy by induction hypothesis and T-INVK.

Case RC-NEW-ARG: Easy by induction hypothesis and T-NEW.

□

THEOREM 2 PROGRESS. *Let* `e` *be a well-typed expression. 1. If* `e` *has* `new C<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`).f` *as a subexpression, then* $\emptyset\vdash$*fields*`(C<`$\overline{\texttt{T}}$`>`$)=\overline{\texttt{U}}\ \overline{\texttt{f}}$, *and* `f = f`$_i$. *2. If* `e` *has* `new C<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`).m(`$\overline{\texttt{d}}$`)` *as a subexpression, then* *mbody*`(m, C<`$\overline{\texttt{T}}$`>`$)=(\overline{\texttt{x}},\texttt{e}_0)$ *and* $|\overline{\texttt{x}}| = |\overline{\texttt{d}}|$.

PROOF. 1. It follows easily from T-FIELD and the well-typedness of subexpressions.

2. Also using well-typedness of subexpression and T-INVK, we have $\Delta;\Lambda\vdash$*mtype*`(m, C<`$\overline{\texttt{T}}$`>`$)=\overline{\texttt{U}}\to$`U`$_0$. It is then easy using the MB-* rules to show that $\Delta;\Lambda\vdash$*mbody*`(m, C<`$\overline{\texttt{T}}$`>`$)=(\overline{\texttt{x}},\texttt{e}_0)$, since MB-* and MT-* rules have a one-to-one correspondence for non-variable types `C<`$\overline{\texttt{T}}$`>`. □

THEOREM 3 TYPE SOUNDNESS. *If* $\emptyset;\emptyset;\emptyset\vdash$`e`$\in$`T` *and* `e`$\longrightarrow^*$`e`$'$, *then* `e`$'$ *is a value* `v` *such that* $\emptyset;\emptyset;\emptyset\vdash$`v`$\in$`S` *and* $\emptyset\vdash$`S`$<:$`T` *for some type* `S`.

PROOF. Conclusion follows from Theorem 1 and Theorem 2 □

LEMMA 1. *If* $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=\overline{\mathtt{S}}\to\mathtt{S}$, *mbody*$(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=(\overline{\mathtt{x}},\mathtt{e}_0)$, *where* $\Delta\vdash\mathtt{C}\langle\overline{\mathtt{T}}\rangle$ *ok, then there exists a type* $\mathtt{S}'$ *such that* $\Delta\vdash\mathtt{S}'<:\mathtt{S}$, $\Delta\vdash\mathtt{S}'$ *ok, and* $\Delta;\Lambda;\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C}\langle\overline{\mathtt{T}}\rangle\vdash\mathtt{e}_0\in\mathtt{S}'$.

PROOF. By induction on the derivation of *mbody*$(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=(\overline{\mathtt{x}},\mathtt{e}_0)$:

Case MB-CLASS-S:

$CT(\mathtt{C})=\mathtt{class\ C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{N}\ \{\ldots\overline{\mathtt{M}}\}\quad \mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\in\overline{\mathtt{M}}\quad \mathtt{e}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}$

By MT-CLASS-S,

$\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}\to\mathtt{U}_0)$

By WF-CLASS, $\Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By T-METH-S,

$\overline{\mathtt{X}}<:\overline{\mathtt{N}};\emptyset;\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C}\langle\overline{\mathtt{X}}\rangle\vdash\mathtt{e}\in\mathtt{U}_0'\quad \overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{U}_0'<:\mathtt{U}_0$

By Lemma 8 and 5, $\Delta;\Lambda;[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C}\langle\overline{\mathtt{X}}\rangle)\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'$

By Lemma 6 and 5, $\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0$.

Let $\overline{\mathtt{S}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}$, $\mathtt{S}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0$, $\mathtt{S}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'$.

Case MB-CLASS-R:

$CT(\mathtt{C})=\mathtt{class\ C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{T}\ \{\ \ldots\ \ \overline{\mathfrak{M}}\}$

$\langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\rangle\mathtt{for}(\mathbb{M}_p;o\mathbb{M}_f)\ \mathtt{S}''\ \eta\ (\overline{\mathtt{S}}''\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e}_0';\}\in\overline{\mathfrak{M}}$

$R_p''=range(\mathbb{M}_p,\langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\rangle)\quad R_n''=range(\mathbb{M}_f,\bullet)\quad \Lambda''=\langle R_p'',R_n''\rangle$

$\Delta''=\overline{\mathtt{X}}<:\overline{\mathtt{N}};\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\quad \Lambda_d=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\langle R_p'',R_n''\rangle)\quad \Delta'';\emptyset\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$

$\Delta'';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d\quad \mathtt{e}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0'$

By MT-CLASS-R and Lemma 2 (thus there is only one method $\mathtt{m}$),

$\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}''\to\mathtt{S}'')$

By T-METH-R,

$\Gamma''=\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}}'',\mathtt{this}\mapsto\mathtt{C}\langle\overline{\mathtt{X}}\rangle\quad \Delta'';\Lambda'';\Gamma''\vdash\mathtt{e}_0'\in\mathtt{T}''\quad \Delta''\vdash\mathtt{T}''<:\mathtt{S}''\quad \Delta''\vdash\overline{\mathtt{N}}\ \mathtt{OK}$

By WF-CLASS and Lemma 5, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta''\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$

By Lemma 6 and Lemma 5, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta''\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}''<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}''$

By Lemma 8, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta'';[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'';[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Gamma''\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}_0'\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_0$

By Lemma 20, and the obvious facts that $[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Delta''=\emptyset$, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma''=\Gamma$, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0'\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0$

By Lemma 7, $\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}''<:\mathtt{S}''$

Let $\overline{\mathtt{S}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}''$, $\mathtt{S}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}''$, $\mathtt{S}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}''$.

Case MB-SUPER-S, MB-SUPER-R: Follows from induction hypothesis.

□

LEMMA 2. *Suppose* $\Delta\vdash disjoint(\Lambda_1,\ \Lambda_2)$, *where*

$\Lambda_1=\langle R_{p_1},oR_{n_1}\rangle\quad R_{p_1}=(\mathtt{T}_1,\ \langle\overline{\mathtt{X}}\lhd\overline{\mathtt{Q}}\rangle\overline{\mathtt{U}}\to\mathtt{U}_0)\quad R_{n_1}=(\mathtt{T}_1',\ \overline{\mathtt{U}}'\to\mathtt{U}_0')$

$\Lambda_2=\langle R_{p_2},o'R_{n_2}\rangle\quad R_{p_2}=(\mathtt{T}_2,\ \langle\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\rangle\overline{\mathtt{V}}'\to\mathtt{V}_0)\quad R_{n_2}=(\mathtt{T}_2',\ \overline{\mathtt{V}}'\to\mathtt{V}_0')$

Then for any Λ_3, *if* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_1$, *then there does not exist* $\overline{\mathtt{W}}'$*such that* $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$

PROOF. We prove by contradiction. Let there by such $\overline{\mathtt{W}}'$ where $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$

Let $\Lambda_3=\langle R_{p_3},o''R_{n_3}\rangle$, where

$R_{p_3}=(\mathtt{T}_3,\ \langle\overline{\mathtt{Z}}\lhd\overline{\mathtt{N}}\rangle\overline{\mathtt{S}}\to\mathtt{S}_0)\quad R_{n_3}=(\mathtt{T}_3',\ \langle\overline{\mathtt{Z}}\lhd\overline{\mathtt{N}}\rangle\overline{\mathtt{S}}\to\mathtt{S}_0')$

By DS-Λ, one of the following mutually exclusive range conditions must hold:

$\Delta\vdash+R_{p_1}\otimes+R_{p_2}\quad \Delta\vdash+R_{p_1}\otimes o'R_{n_2}\quad \Delta\vdash+R_{p_2}\otimes oR_{n_2}\quad \Delta\vdash oR_{n_1}\otimes o'R_{n_2}$

By SB-Λ and SB-R, $\Delta\vdash\overline{\mathtt{P}},\overline{\mathtt{Q}},\overline{\mathtt{N}}$ OK

$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_{p_3}\sqsubseteq_R R_{p_1}$ $\quad$ $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash \mathit{unify}(\mathtt{S}_0:\overline{\mathtt{S}},\ \mathtt{U}_0:\overline{\mathtt{U}})$

$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{p_3}\sqsubseteq_R R_{p_2}$ $\quad$ $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{S}_0:\overline{\mathtt{S}},\ \mathtt{V}_0:\overline{\mathtt{V}})$

By UNI,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{S}_0:\overline{\mathtt{S}})=[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{U}_0:\overline{\mathtt{U}})$ $\quad$ for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash \mathtt{W}_i\prec:_{\overline{\mathtt{X}}}\mathtt{X}_i$

$[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}_0:\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0:\overline{\mathtt{V}}$ $\quad$ for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash \mathtt{W}'_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$

Since neither $\overline{\mathtt{X}}$ or $\overline{\mathtt{Y}}$ appear in $\mathtt{S}_0:\overline{\mathtt{S}}$,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{S}_0:\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}_0:\overline{\mathtt{S}}=\mathtt{S}_0:\overline{\mathtt{S}}$ $\quad$ $[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{U}_0:\overline{\mathtt{U}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0:\overline{\mathtt{V}}$

By Lemma 5,

for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash \mathtt{W}_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{X}_i$

for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash \mathtt{W}'_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{Y}_i$,

It follows that,

$\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}};[(\overline{\mathtt{W}}:\overline{\mathtt{W}}')/(\overline{\mathtt{X}}:\overline{\mathtt{Y}})]\vdash \mathit{unify}(\mathtt{U}_0:\overline{\mathtt{U}},\ \mathtt{V}_0:\overline{\mathtt{V}})$

It directly contradicts $\Delta\vdash +R_{p_1}\otimes +R_{p_2}$ Thus, one of the other mutually exclusive range conditions must hold.

By SB-Λ,

$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash \Lambda_3\sqsubseteq_\Lambda \Lambda_1$ $\quad$ implies $\quad$ $o=o''=+$ or $o=o''=-$

$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash \Lambda_3\sqsubseteq_\Lambda \Lambda_2$ $\quad$ implies $\quad$ $o'=o''=+$ or $o'=o''=-$

Thus, there can only be two options for $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash \Lambda_3\sqsubseteq_\Lambda \Lambda_1$ and $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash \Lambda_3\sqsubseteq_\Lambda \Lambda_2$:

$o=o'=o''=+$ or $o=o'=o''=-$

We now analyze these two cases:

Case $o=o'=o''=+$

By SB-Λ and SB-R,

$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_{n_3}\sqsubseteq_R R_{n_1}$ $\quad$ $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash \mathit{unify}(\mathtt{S}'_0:\overline{\mathtt{S}}',\ \mathtt{U}'_0:\overline{\mathtt{U}}')$

$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{p_3}\sqsubseteq_R R_{p_2}$ $\quad$ $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}};[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{S}'_0:\overline{\mathtt{S}}',\ \mathtt{V}'_0:\overline{\mathtt{V}}')$

By UNI,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{S}'_0:\overline{\mathtt{S}}')=[\overline{\mathtt{W}}/\overline{\mathtt{X}}](\mathtt{U}'_0:\overline{\mathtt{U}}')$ $\quad$ for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash \mathtt{W}_i\prec:_{\overline{\mathtt{X}}}\mathtt{X}_i$

$[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}'_0:\overline{\mathtt{S}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$ $\quad$ for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{N}}\vdash \mathtt{W}'_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$

Since neither $\overline{\mathtt{X}}$ or $\overline{\mathtt{Y}}$ appear in $\mathtt{S}'_0:\overline{\mathtt{S}}'$,

$[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{S}'_0:\overline{\mathtt{S}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}'_0:\overline{\mathtt{S}}'=\mathtt{S}'_0:\overline{\mathtt{S}}'$ $\quad$ $[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{U}'_0:\overline{\mathtt{U}}'=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$

By Lemma 5,

for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash \mathtt{W}_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{X}_i$

for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash \mathtt{W}'_i\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{Y}_i$,

It follows that

$\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}};[(\overline{\mathtt{W}}:\overline{\mathtt{W}}')/(\overline{\mathtt{X}}:\overline{\mathtt{Y}})]\vdash \mathit{unify}(\mathtt{U}'_0:\overline{\mathtt{U}}',\ \mathtt{V}'_0:\overline{\mathtt{V}}')$

This directly contradicts $\Delta\vdash +R_{n_1}\otimes +R_{n_2}$

Thus, it must be true that $\Delta\vdash +R_{p_1}\otimes +R_{n_2}$, or $\Delta\vdash +R_{p_2}\otimes +R_{n_1}$.

Assume that $\Delta\vdash +R_{p_1}\otimes +R_{n_2}$. By $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{n_3}\sqsubseteq_R R_{n_2}$, and Lemma 3, $\Delta\vdash +R_{p_1}\otimes +R_{n_3}$, which makes $\Delta\vdash \mathit{disjoint}(\Lambda_1,\ \Lambda_3)$. This contradicts the assumption.

$\Delta\vdash +R_{p_2}\otimes +R_{n_1}$ results in a similar contradiction.

Case $o=o'=o''=-$

By Lemma 3 and $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{n_2}\sqsubseteq_R R_{n_3}$, $\Delta\vdash +R_{p_1}\otimes -R_{n_3}$, which makes $\Delta\vdash \mathit{disjoint}(\Lambda_1,\ \Lambda_3)$. This contradicts the assumption.

Similar contradiction results from $\Delta\vdash +R_{p_2}\otimes -R_{n_1}$.

□

LEMMA 3. *1) Suppose* $\Delta\vdash +R_1\otimes +R_2$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_3\sqsubseteq_R R_2$, *where R3 has no pattern type variables, then* $\Delta\vdash +R_1\otimes +R_3$.

2) Suppose $\Delta\vdash +R_1\otimes -R_2$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_2\sqsubseteq_R R_3$, *where R3 has no pattern type variables, then* $\Delta\vdash +R_1\otimes -R_3$.

PROOF. Let $R_1=(\mathtt{T}_1, \mathtt{<}\overline{\mathtt{X}}\mathtt{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{V}}\mathtt{\to V})$, $R_2=(\mathtt{T}_2, \mathtt{<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\mathtt{\to U})$, $R_3=(\mathtt{T}_3, \overline{\mathtt{S}}\mathtt{\to S})$
1) We prove by contradiction. Suppose $\Delta\not\vdash +R_1\otimes +R_3$
That means for some $\overline{\mathtt{W}}'$, $\Delta,\overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{Q}};[\overline{\mathtt{W}}'/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\mathtt{V:}\overline{\mathtt{V}},\ \mathtt{S:}\overline{\mathtt{S}})$
By definition of SB-R, $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{X}}]\vdash R_3\sqsubseteq_R R_1$
By UNI, and the fact that neither $\overline{\mathtt{X}}$ or $\overline{\mathtt{Y}}$ appear in $\mathtt{S:}\overline{\mathtt{S}}$,
$[\overline{\mathtt{W}}'/\overline{\mathtt{X}}](\ \mathtt{S:}\overline{\mathtt{S}})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\ \mathtt{S:}\overline{\mathtt{S}})\quad [\overline{\mathtt{W}}'/\overline{\mathtt{X}}](\mathtt{V:}\overline{\mathtt{V}})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{U:}\overline{\mathtt{U}})$
Since $\overline{\mathtt{X}}$ do not appear in $\mathtt{U:}\overline{\mathtt{U}}$, and $\overline{\mathtt{Y}}$ do not appear in $\mathtt{V:}\overline{\mathtt{V}}$,
$[\overline{\mathtt{W}}'/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{V:}\overline{\mathtt{V}})=[\overline{\mathtt{W}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{U:}\overline{\mathtt{U}})$
Thus, $\overline{\mathtt{W}}\mathtt{:}\overline{\mathtt{W}}'$ contradicts the condition in ME-1 for $\Delta\vdash +R_1\otimes +R_2$.
2) Proof follows from Lemma 23 and ME-2. □

LEMMA 4 TERM SUBSTITUTION PRESERVES TYPING. *If* $\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}\in\mathtt{T}$, η *does not appear in* $\mathtt{e}$, $\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{d}}\in\overline{\mathtt{S}}$, $\Delta\vdash\overline{\mathtt{S}}\mathtt{<:}\overline{\mathtt{T}}$, *then* $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}\in\mathtt{T}'$, *for some* $\mathtt{T}'$ *where* $\Delta;\Lambda;\Gamma\vdash\mathtt{T}'\mathtt{<:T}$.

PROOF. By induction on the derivation of $\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}\in\mathtt{T}$.

Case T-VAR: $\mathtt{e=x}_i$
$[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{x=d}_i\quad \Delta;\Lambda;\Gamma\vdash\mathtt{d}_i\in\mathtt{S}_i$
Let $\mathtt{T}'=\mathtt{S}_i$

Case T-FIELD: $\mathtt{e=e}_0\mathtt{.f}_i$
By induction hypothesis and T-FIELD,
$\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{T}_0\quad \Delta;\Lambda;\Gamma\vdash\mathtt{T}_0\mathtt{<:T}$
Conclusion follows from Lemma 11.

Case T-INVK: $\mathtt{e=e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)}$
By premesis of the lemma, $\mathtt{n=m}$
By T-INVK:
$\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}_0\in\mathtt{T}_0\qquad \Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$
$\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{m},\ \mathtt{T}_0)=\overline{\mathtt{T}}\mathtt{\to T}\quad \Delta\vdash\overline{\mathtt{S}}\mathtt{<:}\overline{\mathtt{T}}$
By induction hypothesis,
$\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{T'}_0\quad \Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{T}}'$
$\Delta\vdash\mathtt{T'}_0\mathtt{<:T}_0\qquad \Delta\vdash\overline{\mathtt{T}}'\mathtt{<:}\overline{\mathtt{T}}$
By Lemma 19, $\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{m},\ \mathtt{T'}_0)=\overline{\mathtt{T}}\mathtt{\to T}$
By T-INVK, $\Delta;\Lambda\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}](\mathtt{e}_0\mathtt{.m(}\overline{\mathtt{e}}\mathtt{)})\in\mathtt{T}$

Case T-NEW: $\mathtt{e=new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}$
$\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \mathit{ok}\quad \Delta\vdash\mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$
$\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}\quad \Delta\vdash\overline{\mathtt{S}}\mathtt{<:}\overline{\mathtt{U}}$
By induction hypothesis, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{S}}'$, for some $\overline{\mathtt{S}}'$where $\Delta\vdash\overline{\mathtt{S}}'\mathtt{<:}\overline{\mathtt{S}}$.
By S-TRANS, $\Delta\vdash\overline{\mathtt{S}}'\mathtt{<:}\overline{\mathtt{U}}$.
By Lemma 11 and T-NEW, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}](\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)})\in\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$

□

LEMMA 5 WEAKENING. *Suppose* $\Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}}\vdash\overline{\mathtt{N}}$ *ok, none of* $\overline{\mathtt{X}}$ *appears in* Δ*, and* $\Delta\vdash\mathtt{U}$ *ok.*

(*1*) *If* $\Delta\vdash\mathtt{S}{<:}\mathtt{T}$*, then* $\Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}}\vdash\mathtt{S}{<:}\mathtt{T}$.

(*2*) *If* $\Delta\vdash\mathtt{S}$ *ok, then* $\Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}}\vdash\mathtt{S}$ *ok.*

(*3*) *If* $\Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$*, then* $\Delta;\Lambda;\Gamma,\mathtt{x}\in\mathtt{U}\vdash\mathtt{e}\in\mathtt{T}$ *and* $\Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}};\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$.

(*4*) *If* $\Delta;\emptyset;\Gamma\vdash\mathtt{e}\in\mathtt{T}$*, then* $\Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$.

(*5*) *If* $\Delta\vdash\mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{S}$*, then* $\Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}}\vdash\mathtt{T}\prec:_{\overline{\mathtt{Z}}}\mathtt{S}$.

PROOF. (1) Follows by induction on the derivation of subtyping.

(2) Follows by induction on well-formed types rules.

(3) Follows by induction on expression typing rules.

(4) Follows by induction on expression typing rules.

(5) Follows by pattern matching rules.

□

LEMMA 6 TYPE SUBSTITUTION PRESERVES SUBTYPING.
If $\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2\vdash\mathtt{S}{<:}\mathtt{T}$*, and* $\Delta_1\vdash\overline{\mathtt{U}}{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok and none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1*, then*
$\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{S}{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

PROOF. By induction on the derivation of $\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2\vdash\mathtt{S}{<:}\mathtt{T}$

Case S-REFL: Trivial

Case S-VAR:
If $\mathtt{X}\notin\overline{\mathtt{X}}$, then it is trivial.
If $\mathtt{X}\in\overline{\mathtt{X}}$, $bound_\Delta(\mathtt{X}_i)=\mathtt{N}_i$.
$[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{X}_i=\mathtt{U}_i$ $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]bound_\Delta(\mathtt{X}_i)=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$.
By assumption and Lemma 5, $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash\mathtt{U}_i{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$.

Case S-TRANS: By induction hypothesis.

Case S-CLASS: Trivial.

□

LEMMA 7 PATTERN-MATCHING TYPE SUBSTITUTION PRESERVES SUBTYPING.
If $\Delta\vdash\mathtt{S}{<:}\mathtt{T}$*,* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$*,* $\Delta\vdash\overline{\mathtt{Y}}{<:}\overline{\mathtt{P}}$*,* $\Delta\vdash\overline{\mathtt{W}}$ *ok, and* $\overline{\mathtt{Y}}$ *do not appear in* Λ*, then* $\Delta\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}{<:}[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}$.

PROOF. By induction on the derivation of subtyping relation:

Case S-REFL: Trivial.

Case S-VAR:
If $\mathtt{X}\notin\overline{\mathtt{Y}}$, conclusion is thus trivial.
If $\mathtt{X}\in\overline{\mathtt{Y}}$, let $\mathtt{X}=\mathtt{Y}_i$. $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{X}=\mathtt{W}_i$. By Lemma 22, $\Delta\vdash\mathtt{W}_i{<:}[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i$.

Case S-TRANS: By induction hypothesis and S-TRANS.

Case S-CLASS: Easy by S-CLASS.

□

LEMMA 8 TYPE SUBSTITUTION PRESERVES TYPING. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$ *and* $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *where* $\mathtt{e}$ *is fully grounded,* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *and* Λ,
then $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$.

PROOF. By induction on the derivation of $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$
Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2$

Case T-VAR: Trivial

Case T-FIELD: $\Delta_o;\Lambda;\Gamma\vdash\mathtt{e}_0\mathtt{.f}_i\in\mathtt{T}_i$
$\Delta_o;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0$ $\Delta\vdash \mathit{fields}(\mathit{bound}_{\Delta_o}(\mathtt{T}_0))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}$
By induction hypothesis, $\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$
By Lemma 10, $\Delta_n\vdash \mathit{bound}_{\Delta_n}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0)<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_o}(\mathtt{T}_0))$
By Lemma 11,
$\Delta_n\vdash \mathit{fields}(\mathit{bound}_{\Delta_n}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0))=\overline{\mathtt{S}}\ \overline{\mathtt{g}}$,
$\mathtt{f}_j=\mathtt{g}_j$, $\mathtt{S}_j=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_j$ for $j\leq\#(\overline{\mathtt{f}})$.
By T-FIELD, $\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0\mathtt{.f}_i\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_i$.

Case T-INVK: $\Delta_o\vdash\mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{T}$
$\Delta_o;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0$ $\Delta_o;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$
$\Delta_o;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T}_0)=\overline{\mathtt{T}}\to\mathtt{T}$ $\Delta_o\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$
By induction hypothesis,
$\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{e}_0\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0$ $\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{e}}\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$
By Lemma 6, $\Delta_n\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}$
By Lemma 13, and that $\mathtt{e}$ is fully grounded, and thus $\mathtt{n}$ must be $\mathtt{m}$ and not η,
$\Delta_n\vdash \mathit{mtype}(\mathtt{n},\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_0)=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{T}}\to\mathtt{T})$
By T-INVK, $\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)})\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}$

Case T-NEW: $\Delta_o\vdash\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$
$\Delta_o\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ ok$ $\Delta_o\vdash \mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{S}}\ \overline{\mathtt{f}}$
$\Delta_o;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}'$ $\Delta_o\vdash\overline{\mathtt{S}}'<:\overline{\mathtt{S}}$
By Lemma 9, $\Delta_n\vdash\mathtt{C<}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\mathtt{>}\ ok$
By definition of *fields*:
$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{Y}}\mathtt{\lhd}\overline{\mathtt{N}}\mathtt{>\lhd S\ \{}\overline{\mathtt{D}}\ \overline{\mathtt{g}}\ \mathtt{...\}}$ $\Delta_n\vdash \mathit{fields}(\mathit{bound}_{\Delta_o}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}))=\overline{\mathtt{D}}'\ \overline{\mathtt{g}}'$
$\overline{\mathtt{S}}\ \overline{\mathtt{f}}=[\overline{\mathtt{T}}/\overline{\mathtt{Y}}](\overline{\mathtt{D}}\ \overline{\mathtt{g}},\ \overline{\mathtt{D}}'\ \overline{\mathtt{g}}')$
Since $\overline{\mathtt{X}}$ cannot appear in $\mathtt{S}$, by the definition of *bound*,
$\mathit{bound}_{\Delta_o}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})=\mathit{bound}_{\Delta_n}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})=[\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}$
By Lemma 12, $\Delta_n\vdash \mathit{fields}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})=\overline{\mathtt{D}}'\ \overline{\mathtt{g}}'$
It follows that, $\Delta_n\vdash \mathit{fields}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}$.
By Lemma 8 and 6,
$\Delta_n;\Lambda;[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Gamma\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{e}}\in[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}'$ $\Delta_n\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}'<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}$
The conclusion follows from T-NEW.

□

LEMMA 9. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\mathtt{T}\ ok$, *and* $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1\vdash\overline{\mathtt{U}}\ ok$, *none of* $\overline{\mathtt{X}}$ *appearing in* Δ_1, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}\ ok$.

PROOF. By induction on the derivation of $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash\mathtt{T}\ ok$

Case WF-OBJECT: Trivial.

Case WF-VAR: If $\mathtt{X}$ not in $\overline{\mathtt{X}}$, then it's trivial. Otherwise, $\mathtt{X}{=}\mathtt{X}_i$, $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{X}_i{=}\mathtt{U}_i$, by assumption, $\Delta_1{\vdash}\mathtt{U}_i$ *ok*. And by Lemma 5, conclusion follows.

Case WF-CLASS: By induction hypothesis and Lemma 6.

□

LEMMA 10. *Suppose* $\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2{\vdash}\mathtt{T}$*ok, and* $\Delta_1{\vdash}\overline{\mathtt{U}}{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$ *with* $\Delta_1{\vdash}\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1*. Then,*
$\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2{\vdash}\mathit{bound}_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}){<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}(\mathtt{T}))$

PROOF. If $\mathtt{T}$ is a non-variable type, then the conclusion is trivial.
If $\mathtt{T}$ is a variable type, but $\mathtt{T}{\notin}\overline{\mathtt{X}}$, then the conclusion is also trivial.
If $\mathtt{T}{\in}\overline{\mathtt{X}}$,
$\mathit{bound}_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}){=}\mathtt{U}_i \quad [\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}(\mathtt{T})){=}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{N}_i$
By assumption and Lemma 5, conclusion follows.

□

LEMMA 11.
If $\Delta{\vdash}\mathtt{S}{<:}\mathtt{T}$*, and* $\Delta{\vdash}\mathit{fields}(\mathit{bound}_\Delta(\mathtt{T})){=}\overline{\mathtt{T}}\ \overline{\mathtt{f}}$*, then* $\Delta{\vdash}\mathit{fields}(\mathit{bound}_\Delta(\mathtt{S})){=}\overline{\mathtt{S}}\ \overline{\mathtt{g}}$*, and* $\mathtt{S}_i{=}\mathtt{T}_i$*, and* $\mathtt{g}_i{=}\mathtt{f}_i$ *for all* $i \le \#(\overline{\mathtt{f}})$*.*

PROOF. By induction on the derivation of $\Delta{\vdash}\mathtt{S}{<:}\mathtt{T}$.

Case S-REFL: Trivial.

Case S-VAR: $\mathit{bound}_\Delta(\mathtt{S}){=}\mathit{bound}_\Delta(\mathtt{T})$. Conclusion follows.

Case S-TRANS: By induction hypothesis.

Case S-CLASS: $\Delta{\vdash}\mathtt{C}{<}\overline{\mathtt{T}}{>}{<:}[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}$
By FD-CLASS,
$\mathtt{class\ C}{<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{N}}{>}{\lhd}\mathtt{T}\ \{\overline{\mathtt{U}}\ \overline{\mathtt{f}}\ \mathtt{...}\} \qquad \Delta{\vdash}\mathit{fields}(\mathit{bound}_\Delta([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})){=}\overline{\mathtt{T}}\ \overline{\mathtt{f}}$
$\Delta{\vdash}\mathit{fields}(\mathtt{C}{<}\overline{\mathtt{T}}{>}){=}\overline{\mathtt{T}}\ \overline{\mathtt{f}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}\ \overline{\mathtt{f}}$
Conclusion is obvious.

□

LEMMA 12. *If* $\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2{\vdash}\mathit{fields}(\mathtt{T}){=}\overline{\mathtt{S}}\ \overline{\mathtt{f}}$ $\Delta_1{\vdash}\overline{\mathtt{U}}{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$*, where* $\Delta_1{\vdash}\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1*, then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2{\vdash}\mathit{fields}([\overline{\mathtt{U}}/\overline{\mathtt{T}}]\mathtt{T}){=}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}},\overline{\mathtt{S}'}\ \overline{\mathtt{f}'}$*, for some* $\overline{\mathtt{S}'}$ *and* $\overline{\mathtt{f}'}$*.*

PROOF. We prove by case analysis on the definition of *fields*.

Case FD-OBJ: $\mathtt{T}{=}\mathtt{Object}$. Trivial.

Case FD-CLASS: $\mathtt{T}{=}\mathtt{C}{<}\overline{\mathtt{T}}{>}$
$CT(\mathtt{C}){=}\mathtt{class\ C}{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{N}}{>}{\lhd}\mathtt{S}\ \{\ \overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \}$
$\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{U}},\Delta_2{\vdash}\mathit{fields}(\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{U}},\Delta_2}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S})){=}\overline{\mathtt{D}}\ \overline{\mathtt{g}}$
By induction hypothesis, for some $\overline{\mathtt{D}'}$, $\overline{\mathtt{g}'}$,
$\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2{\vdash}\mathit{fields}([\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}))){=}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{D}}\ \overline{\mathtt{g}}, \overline{\mathtt{D}'}\ \overline{\mathtt{g}'}$.
By Lemma 10,
$\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2{\vdash}\mathit{bound}_{\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2}([\overline{\mathtt{U}}/\overline{\mathtt{X}}][\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}){<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\mathit{bound}_{\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2}([\overline{\mathtt{T}}/\overline{\mathtt{Y}}]\mathtt{S}))$
Conclusion then follows from Lemma 11.

□

LEMMA 13. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{V}}\to\mathtt{V}_0$, $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *or* Λ, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T})=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$.

PROOF. By induction on the derivation of $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{V}}\to\mathtt{V}_0$.
Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2$

Case MT-VAR-S: $\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{X})=\overline{\mathtt{V}}\to\mathtt{V}_0$
If $\mathtt{X}$ not in $\overline{\mathtt{X}}$, easy.
If $\mathtt{X}$ is in $\overline{\mathtt{X}}$, let $[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{X}=\mathtt{U}_i$
$\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ bound_{\Delta_o}(\mathtt{X}))=\overline{\mathtt{V}}\to\mathtt{V}_0$
By induction hypothesis, $\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{U}}/\overline{\mathtt{X}}]bound_{\Delta_o}(\mathtt{X}))=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$
By Lemma 10, $\Delta_n\vdash bound_{\Delta_n}(\mathtt{U}_i)<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}](bound_{\Delta_o}(\mathtt{X}))$.
By Lemma 19 $\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ bound_{\Delta_n}(\mathtt{U}_i))=[\overline{\mathtt{U}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$
By MT-VAR-S again, the conclusion follows.

Case MT-CLASS-S, MT-SUPER-S: Trivial through type substitutions.

Case MT-CLASS-R:
$\Delta_o;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r \quad \Delta_o;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$
By Lemma 14,
$\Delta_n;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda'_r \quad \Delta_n;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda'_r\sqsubseteq_\Lambda\Lambda_d$
for some $\overline{\mathtt{W}}'$. Conclusion follows by MT-CLASS-R.

Case MT-SUPER-R:
$CT(\mathtt{C})=\mathtt{class\ C}<\overline{\mathtt{Y}}\lhd\overline{\mathtt{N}}>\lhd\mathtt{T}\ \{\ldots\ \ \overline{\mathfrak{M}}\}$
for all $\mathfrak{M}_i\in\overline{\mathfrak{M}}$, $\Lambda_d=reflectiveEnv(\mathfrak{M}_i)$
$\Delta_o;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r \quad \Delta_o\vdash disjoint(\Lambda_r,\ \Lambda_d)$
By Lemma 15,
$\Delta_n;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda'_r \quad \Delta_n\vdash disjoint(\Lambda'_r,\ \Lambda_d)$
Conclusion follows from induction hypothesis.

□

LEMMA 14. *If*
$\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$, $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$, $\Delta_1\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{T}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *or* Λ, *then*
$\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda'{}_r$, *and* $\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda'{}_r\sqsubseteq_\Lambda\Lambda_d$, *where* $\overline{\mathtt{W}}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.

PROOF. Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$
By the definition of *specialize*,
$\Lambda_d=\langle R_p,oR_n\rangle \qquad R_p=(\mathtt{T}_i,\ <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U}) \quad R_n=(\mathtt{T}_j,\ \overline{\mathtt{V}}\to\mathtt{V})$
$\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_i)=\overline{\mathtt{U}}'\to\mathtt{U}' \quad R'_p=(\mathtt{T}_i,\ \overline{\mathtt{U}}'\to\mathtt{U}')$
By Lemma 13, $\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}'\to\mathtt{U}')$
Let $R''_p=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}'\to\mathtt{U}')$
By Lemma 16, $\Delta_n;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R''_p\sqsubseteq_R R_p$, where $\overline{\mathtt{W}}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.

Case $o=+$, $\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_j)=\overline{\mathtt{V}}'\to\mathtt{V}'_0$,
$R'_n=(\mathtt{T}_j,\ \overline{\mathtt{V}}'\to\mathtt{V}_0)$
By Lemma 13,

$\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}')$
Let $R''_n=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}'))$
By Lemma 16, $\Delta_n;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash R''_n\sqsubseteq_R R_n$.

Case o=-,
$R'_n=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R_n$.
Let $R''_n=[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]R_n$, clearly, $\Delta_n;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash R_n\sqsubseteq_R R''_n$.
Since none of $\overline{\mathtt{X}}$ appears in Λ, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i=\mathtt{T}_i$, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j=\mathtt{T}_j$.
Then let $\Delta_n;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\langle R''_p,R''_n\rangle$. By SB-$\Lambda$, $\Delta_n;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash\langle R''_p,R''_n\rangle\sqsubseteq_\Lambda\Lambda_d$.

□

Lemma 15. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$, $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2\vdash disjoint(\Lambda_r,\ \Lambda_d)$, $\Delta_1\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{T}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1 *or* Λ, *then* $\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda'_r$, *and* $\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2\vdash disjoint(\Lambda'_p,\ \Lambda_d)$.

Proof. Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$
By the definition of *specialize*,
$\Lambda_d=\langle R_p,oR_n\rangle$ $R_p=(\mathtt{T}_i,\ <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U})$ $R_n=(\mathtt{T}_j,\ \overline{\mathtt{V}}\to\mathtt{V})$
$\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_i)=\overline{\mathtt{U}'}\to\mathtt{U}'$ $R'_p=(\mathtt{T}_i,\ \overline{\mathtt{U}'}\to\mathtt{U}')$
By Lemma 13,
$\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}'}\to\mathtt{U}')$
Let $R''_p=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}'}\to\mathtt{U}')$

Case o=+, $\Delta_o;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_j)=\overline{\mathtt{V}'}\to\mathtt{V}'_0$,
$R'_n=(\mathtt{T}_j,\ \overline{\mathtt{V}'}\to\mathtt{V}_0)$
By Lemma 13,
$\Delta_n;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}')$
Let $R''_n=([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}'}\to\mathtt{V}'))$

Case o=-,
$R'_n=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R_n$.
Let $R''_n=[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]R_n$, clearly, $\Delta_n;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash R_n\sqsubseteq_R R''_n$.
Since none of $\overline{\mathtt{X}}$ appears in Λ, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_i=\mathtt{T}_i$, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_j=\mathtt{T}_j$. The let $\Delta_n;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\langle R''_p,R''_n\rangle$.
The by Lemma 18 and DS-Λ, if there was originally a pair of mutually exclusive range conditions in Λ_r, then there is a pair of mutually exclusive range conditions in Λ'_r.

□

Lemma 16 Substitution Preserves Single Range Containment. *If* $\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$, $\Delta_1\vdash\overline{\mathtt{U}}<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1\vdash\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1, *none of* $\overline{\mathtt{X}}$ *appears on* $\overline{\mathtt{Y}}$, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$, *where* $\overline{\mathtt{W}'}=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{W}}$.

Proof. Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$
By SB-R,
$R_1=(\mathtt{T}_1,\ <\overline{\mathtt{X}}\lhd\overline{\mathtt{Q}}>\overline{\mathtt{U}}\to\mathtt{U}_0)$ $R_2=(\mathtt{T}_2,\ \overline{\mathtt{V}}\to\mathtt{V}_0)$
$\Delta_o\vdash\mathtt{T}_2<:\mathtt{T}_1$ $\Delta'_o=\Delta_o,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}$ $\Delta'_o;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash unify(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})$
By Lemma 6, $\Delta_n\vdash[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_2<:[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{T}_1$

By UNI, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{U}_0{:}\overline{\mathtt{U}}=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{V}_0{:}\overline{\mathtt{V}}$, for all $\mathtt{Y}_i{\in}\overline{\mathtt{Y}}, \Delta_o{\vdash}\mathtt{W}_i{\prec:}_{\overline{\mathtt{Y}}}\mathtt{Y}_i$
Clearly, $[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{U}_0{:}\overline{\mathtt{U}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0{:}\overline{\mathtt{V}}$
By Lemma 17, $\Delta_n{\vdash}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}_i{\prec:}_{\overline{\mathtt{Y}}}\mathtt{Y}_i$ for all $\mathtt{W}_i$.
It follows that $\Delta_n;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]{\vdash}\mathit{unify}([\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{U}_0{:}\overline{\mathtt{U}}, [\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{V}_0{:}\overline{\mathtt{V}})$
Conclusion follows from SB-R. □

Lemma 17 Substitution Preserves Pattern Type Pattern Match. *If* $\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2{\vdash}\mathtt{W}{\prec:}_{\overline{\mathtt{Y}}}\mathtt{Y}$, *where* $\Delta_1{\vdash}\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1, *none of* $\overline{\mathtt{X}}$ *appears on* $\overline{\mathtt{Y}}$, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2{\vdash}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W}{\prec:}_{\overline{\mathtt{Y}}}\mathtt{Y}$.

Proof. We prove by induction on pattern matching rules PM-*.

Case PM-REFL: Trivial

Case PM-CL: By induction hypothesis.

Case PM-CL-S: By induction hypothesis.

Case PM-VAR: By inspection of definition of *bound*, $\mathit{bound}_\Delta([\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathtt{W})=[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\mathit{bound}_\Delta(\mathtt{W})$. Conclusion follows by induction hypothesis.

Case PM-PVARS: By induction hypothesis and inspection of definition of *bound*, similar to above case.

□

Lemma 18 Substitution Preserves Range Mutual Exclusion. *If* $\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2{\vdash}oR_1{\otimes}o'R_2$, $\Delta_1{\vdash}\overline{\mathtt{U}}{<:}[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}$, *where* $\Delta_1{\vdash}\overline{\mathtt{U}}$ *ok, and none of* $\overline{\mathtt{X}}$ *appears in* Δ_1,R_1, *or* R_2, *then* $\Delta_1,[\overline{\mathtt{U}}/\overline{\mathtt{X}}]\Delta_2{\vdash}oR_1{\otimes}o'R_2$.

Proof. Let $\Delta_o=\Delta_1,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\Delta_2$, $\Delta_n=\Delta_1,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta_2$
We prove by case analysis of ME-1 and ME-2.

Case ME-1:
$R_1=(\mathtt{T}, {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U}_0)$ $R_2=(\mathtt{S}, {<}\overline{\mathtt{Z}}{\lhd}\overline{\mathtt{Q}}{>}\overline{\mathtt{V}}{\to}\mathtt{V}_0)$
$\Delta_o{\vdash}\mathtt{T}{<:}\mathtt{S}$ or $\mathtt{S}{<:}\mathtt{T}$ $\Delta_o'=\Delta_o,\overline{\mathtt{Y}}{<:}\overline{\mathtt{P}},\overline{\mathtt{Z}}{<:}\overline{\mathtt{Q}}$
For all $\overline{\mathtt{W}}$, $\Delta_o';[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]{\vdash}\mathit{unify}(\overline{\mathtt{U}}, \overline{\mathtt{V}})$ implies $[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]\mathtt{U}_0{\neq}[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]\mathtt{V}_0$
Since $\overline{\mathtt{X}}$ do not appear in R_1 or R_2, by Lemma 6, $\Delta_n{\vdash}\mathtt{T}{<:}\mathtt{S}$ or $\mathtt{S}{<:}\mathtt{T}$.
By definition of UNI and the PM-* rules,
for all $\overline{\mathtt{W}}$, $\Delta_n';[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]{\vdash}\mathit{unify}(\overline{\mathtt{U}}, \overline{\mathtt{V}})$ implies $[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]\mathtt{U}_0{\neq}[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]\mathtt{V}_0$

Case ME-2:
$R_1=(\mathtt{T}, {<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}{>}\overline{\mathtt{U}}{\to}\mathtt{U}_0)$ $R_2=(\mathtt{S}, {<}\overline{\mathtt{Z}}{\lhd}\overline{\mathtt{Q}}{>}\overline{\mathtt{V}}{\to}\mathtt{V}_0)$
$\Delta_o'=\Delta_o,\overline{\mathtt{Y}}{<:}\overline{\mathtt{P}},\overline{\mathtt{Z}}{<:}\overline{\mathtt{Q}}$ $\Delta_o';[\overline{\mathtt{W}}/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]{\vdash}R_1{\sqsubseteq}_R R_2$
By Lemma 16 and the fact that $\overline{\mathtt{X}}$ do not appear in R_1 or R_2, $\Delta_n,\overline{\mathtt{Y}}{<:}\overline{\mathtt{P}},\overline{\mathtt{Z}}{<:}\overline{\mathtt{Q}}{\vdash}R_1{\sqsubseteq}_R R_2$.

□

Lemma 19. *If* $\Delta;\Lambda{\vdash}\mathit{mtype}(\mathtt{m}, \mathtt{T})=\overline{\mathtt{U}}{\to}\mathtt{U}_0$, $\Delta{\vdash}\mathtt{S}{<:}\mathtt{T}$, *then* $\Delta;\Lambda{\vdash}\mathit{mtype}(\mathtt{m}, \mathtt{S})=\overline{\mathtt{U}}{\to}\mathtt{U}_0$.

Proof. By induction on the derivation of $\Delta{\vdash}\mathtt{S}{<:}\mathtt{T}$

Case S-REFL: Trivial.

Case S-VAR: $\Delta\vdash \mathtt{X}<:\Delta(\mathtt{X})$

MT-VAR-R1 and MT-VAR-R2 do not apply, since $\mathtt{m}\neq\eta$.

By definition, $bound_\Delta(\mathtt{X})=\Delta(\mathtt{X})$. Conclusion follows from MT-VAR-S.

Case S-TRANS: Easy by induction hypothesis.

Case S-CLASS: $\mathtt{T}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}$, $\mathtt{S}=\mathtt{C}\langle\overline{\mathtt{T}}\rangle$

$\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{C}\langle\overline{\mathtt{T}}\rangle)=\overline{\mathtt{V}}\to\mathtt{V}_0$ can oly be retrieved via MT-CLASS-* and MT-SUPER-*.

If retrieved using MT-CLASS-R:

$CT(\mathtt{C})=\mathtt{class\ C}\langle\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\rangle\lhd\mathtt{T}\ \{\ \ldots\ \ \overline{\mathfrak{M}}\ \}$

$\mathfrak{M}_i=¡\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}¿\mathtt{for}(\mathbb{M}_p\,;o\mathbb{M}_f)\ \mathtt{S}_0\ \eta\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\ \mathfrak{M}_i\in\overline{\mathfrak{M}}$

$\mathbb{M}_p=\mathtt{U}_0\ \eta\ (\overline{\mathtt{U}})\ :\ \mathtt{X}_i\mathtt{.methods}$ $\quad\mathbb{M}_f=\mathtt{U'}_0\ \eta\ (\overline{\mathtt{U}'})\ :\ \mathtt{X}_j\mathtt{.methods}$

$R_p=(\mathtt{X}_i,\ \overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}(\overline{\mathtt{U}}\to\mathtt{U}_0))$ $\quad R_n=(\mathtt{X}_j,\ \overline{\mathtt{U}'}\to\mathtt{U'}_0)$

$\Lambda'=\langle R_p,oR_n\rangle$ $\quad\Lambda_d=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'$

$\Delta;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$ $\quad\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$

$\overline{\mathtt{V}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}$ $\quad\mathtt{V}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}_0$

By T-METH-R and the definition of *override*,

$\Delta'=\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}$ $\quad\Delta'\vdash\overline{\mathtt{N}},\overline{\mathtt{P}}$ ok $\quad\Delta';\Lambda'\vdash override(\eta,\ \mathtt{T},\ \overline{\mathtt{S}}\to\mathtt{S}_0)$

$\Delta';\Lambda'\vdash mtype(\eta,\ \mathtt{T})=\overline{\mathtt{S}}\to\mathtt{S}_0$ $\quad\Delta'\vdash validRange(\Lambda',\ \mathtt{T})$

What we need to show now is that in all cases of MT-*,

$\Delta';\Lambda'\vdash mtype(\eta,\ \mathtt{T})=\overline{\mathtt{S}}\to\mathtt{S}_0$ implies $\Delta;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\to\mathtt{S}_0)$

—MT-VAR-R1: $\Delta';\Lambda'\vdash mtype(\eta,\ \mathtt{X}_i)=\overline{\mathtt{S}}\to\mathtt{S}_0$

By definition of *specialize*,

$\Lambda_d=\langle[\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_p,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_n\rangle$ $\quad[\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_p=(\mathtt{T}_i,\ \overline{\mathtt{Y}}\lhd[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{P}}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}\to[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0))$

$\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_i)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$

Since $\overline{\mathtt{Y}}$ do not appear in $\mathtt{V}_0$ or $\overline{\mathtt{V}}$, $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_i)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{V}}\to\mathtt{V}_0)$.

By $\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda_d$, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\mathtt{S}_0{:}\overline{\mathtt{S}})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{V}_0{:}\overline{\mathtt{V}})$

—MT-VAR-R2: Using similar technique to MT-VAR-R1. The information we use from the definition of *specialize* is when o=+, and $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T}_j)$ is similarly defined.

—MT-VAR-S: By the fact that if $\mathtt{X}$ is not the reflective type of R_p, then $\mathtt{T}_i$ is not the reflective type of $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]R_p$, either. And conclusion follows from induction hypothesis.

—MT-CLASS-R: $\Delta';\Lambda'\vdash mtype(\eta,\ \mathtt{D}\langle\overline{\mathtt{Q}}\rangle)=\overline{\mathtt{S}}\to\mathtt{S}_0$.

$CT(\mathtt{D})=\mathtt{class\ D}\langle\overline{\mathtt{Z}}\lhd\overline{\mathtt{R}}\rangle\lhd\mathtt{R}\ \{\ \ldots\ \ \overline{\mathfrak{M}}\}$

$\mathfrak{M}_i\in\overline{\mathfrak{M}}$ $\ \mathfrak{M}_i=\langle\overline{\mathtt{Y}'}\lhd\overline{\mathtt{P}'}\rangle\mathtt{for\ S'}_0\ \eta\ (\overline{\mathtt{S}'}\ \overline{\mathtt{x}})\ \{\ \ldots\}$

$\Lambda_d{}'=[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}](reflectiveEnv(\mathfrak{M}_i))$ $\quad\Delta';[\overline{\mathtt{W}'}/\overline{\mathtt{Y}'}]\vdash\Lambda'\sqsubseteq_\Lambda\Lambda_d{}'$

$\overline{\mathtt{S}}=[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}][\overline{\mathtt{W}'}/\overline{\mathtt{Y}'}]\overline{\mathtt{S}'}$ $\quad\mathtt{S}_0=[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}][\overline{\mathtt{W}'}/\overline{\mathtt{Y}'}]\mathtt{S}_0{}'$

By Lemma 16 and Lemma 5, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta';[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}'}/\overline{\mathtt{Y}'}]\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'\sqsubseteq_\Lambda[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'_d$

By Lemma 23, $\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Delta';[\overline{\mathtt{W}'}/\overline{\mathtt{Y}'}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda_d{}'$

Since $\overline{\mathtt{Y}}$ do not appear anywhere in Λ_r, or $\Lambda_d{}'$, $\Delta;[\overline{\mathtt{W}'}/\overline{\mathtt{Y}'}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda_d{}'$

It follows from MT-CLASS-R that $\Delta;\Lambda\vdash mtype(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{D}\langle\overline{\mathtt{Q}}\rangle)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\to\mathtt{S}_0)$

—MT-SUPER-R: By induction hypothesis.

—MT-CLASS-S and MT-SUPER-S: Do not apply.

If retrieved using MT-SUPER-S or MT-SUPER-R, then the conclusion is obvious from these definitions.

□

LEMMA 20. *If* $\Delta;\Lambda\vdash \mathit{specialize}(\mathtt{m}, \Lambda_d)=\Lambda_r$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$, *and* $\Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$. *Then* $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}$.

PROOF. By induction on the derivation of $\Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$

Case T-VAR: Trivial.

Case T-FIELD: $\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\mathtt{.f}_i\in\mathtt{T}_i$.
$\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0$ $\Delta\vdash \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}$
By induction hypothesis, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0$
By Lemma 11 and Lemma 5, $\Delta\vdash \mathit{fields}([\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0)=[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\overline{\mathtt{T}}\ \overline{\mathtt{f}}, \overline{\mathtt{S}}\ \overline{\mathtt{g}}$.
By T-FIELD, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0\mathtt{.f}_i\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_i$

Case T-INVK: $\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{T}$
$\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0$ $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n}, \mathtt{T}_0)=\overline{\mathtt{T}}\to\mathtt{T}$ $\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$ $\Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$
By induction hypothesis, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\overline{\mathtt{e}}\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}$
By Lemma 7, $\Delta\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{T}}$.
Also by induction hypothesis, $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\mathtt{e}_0\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0$
If $\mathtt{n}=\mathtt{m}$, or $\mathtt{n}=\eta$, $[\mathtt{m}/\eta]\mathtt{n}=\mathtt{m}$, by Lemma 21, $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{T}}\to\mathtt{T})$.
If $\mathtt{n}=\mathtt{m}'$, where $\mathtt{m}'\neq\mathtt{m}$, $\overline{\mathtt{Y}}$ do not appear in $\overline{\mathtt{T}}$ or $\mathtt{T}$, thus,
$\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash \mathit{mtype}(\mathtt{m}', [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{T}}\to\mathtt{T})$
In either case, it follows from T-INVK that $\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta](\mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)})\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}$.

Case T-NEW: $\Delta;\Lambda;\Gamma\vdash\mathtt{new\ C<}\overline{\mathtt{T}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$
$\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ ok $\Delta\vdash \mathit{fields}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{U}}\ \overline{\mathtt{f}}$ $\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$ $\Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$
By induction hypothesis and Lemma 7,
$\Delta;\Lambda;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\Gamma\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\eta]\overline{\mathtt{e}}\in[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}$ $\Delta\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$
By definition of *fields* and the fact that its definition does not involve any pattern matching variables, $\Delta\vdash \mathit{fields}([\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\ \overline{\mathtt{f}}$
Conclusion follows from T-NEW.

□

LEMMA 21. *Suppose* $\Delta;\Lambda\vdash \mathit{specialize}(\mathtt{m}, \Lambda_d)=\Lambda_r$, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$. *If* $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, \mathtt{T})=\overline{\mathtt{S}}\to\mathtt{S}$, *then* $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T})=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\to\mathtt{S})$.

PROOF. By induction on the derivation of $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, \mathtt{T})=\overline{\mathtt{S}}\to\mathtt{S}$:

Case MT-VAR-R1 and MT-VAR-R2 do not apply.

Case MT-VAR-S: $\mathtt{T}=\mathtt{X}$.
If $\mathtt{X}\notin\overline{\mathtt{Y}}$, then $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{X}=\mathtt{X}$. It is also impossible from method typing rules for $\overline{\mathtt{Y}}$ to appear in the method definitions of $\mathit{bound}_\Delta(\mathtt{X})$, or its method types. The conclusion follows naturally.
If $\mathtt{X}\in\overline{\mathtt{Y}}$, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{X}=\mathtt{W}_i$, $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{Y}_i))=\overline{\mathtt{S}}\to\mathtt{S}$, $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, \mathtt{Y}_i)=\overline{\mathtt{S}}\to\mathtt{S}$.
It follows from induction hypothesis and $\mathit{bound}_\Delta(\mathtt{Y}_i)=\mathtt{P}_i$ that,
$\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\to\mathtt{S})$.
By Lemma 22, $\Delta\vdash\mathtt{W}_i<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i$.
By Lemma 19, $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m}, \mathtt{W}_i)=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}\to\mathtt{S})$.

Case MT-CLASS-S, MT-SUPER-S: Eas following type substitution.

Case MT-CLASS-R: Conclusion follows easily from the definition of *specialize*.

Case MT-SUPER-R: By induction hypothesis.

□

LEMMA 22 UNIFICATION MAPPING PRESERVES TYPE VARIABLE BOUNDS. *If* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$, *and* $\overline{\mathtt{Y}}$ *do not appear anywhere in* Λ, $\Lambda'=\langle R'_p,oR'_n\rangle$, *where* $R'_p=(\mathtt{T},\ <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{U}}\to\mathtt{U})$, *then* $\Delta\vdash\mathtt{W}_i<:[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{P}_i$, *for all* $\mathtt{W}_i\in\overline{\mathtt{W}}$.

PROOF. Let $\Lambda=\langle R_p,oR_n\rangle$, where $R_p=(\mathtt{S},\ \overline{\mathtt{V}}\to\mathtt{V})$, $\Delta'=\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}}$.
By SB-R, $\Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash unify(\mathtt{U}:\overline{\mathtt{U}},\ \mathtt{V}:\overline{\mathtt{V}})$.
By UNI, $\Delta'\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$ for all $\mathtt{W}_i\in\overline{\mathtt{W}}$. We inspect the rules of PM-*:
Since $\overline{\mathtt{Y}}$ cannot appear in $\overline{\mathtt{W}}$, the first rule that applies is PM-VAR:
$bound_{\Delta'}(\mathtt{W}_i)=\mathtt{C}<\overline{\mathtt{T}}>$ $\Delta'\vdash\mathtt{C}<\overline{\mathtt{T}}>\prec:_{\overline{\mathtt{Y}}}[\mathtt{C}<\overline{\mathtt{T}}>/\mathtt{Y}_i]bound_\Delta(\mathtt{Y}_i)$, where $bound_\Delta(\mathtt{Y}_i)=\mathtt{P}_i$
We next prove by derivation of $\Delta'\vdash\mathtt{C}<\overline{\mathtt{T}}>\prec:_{\overline{\mathtt{Y}}}[\mathtt{C}<\overline{\mathtt{T}}>/\mathtt{Y}_i]\mathtt{P}_i$

Case PM-REFL: $[\mathtt{C}<\overline{\mathtt{T}}>/\mathtt{Y}_i]\mathtt{P}_i=\mathtt{C}<\overline{\mathtt{T}}>=bound_{\Delta'}(\mathtt{W}_i)$
Since $\overline{\mathtt{Y}}$ do not appear in Δ and consequently, $\mathtt{C}<\overline{\mathtt{T}}>$, $\Delta\vdash\mathtt{W}_i<:\mathtt{C}<\overline{\mathtt{T}}>$.

Case PM-CL: Let $\mathtt{P}_i=\mathtt{C}<\overline{\mathtt{S}}>$. Since $\mathtt{Y}_i$ do not appear in $\overline{\mathtt{S}}$ after substitution, for $\overline{\mathtt{T}}\prec:_{\overline{\mathtt{Y}}}\overline{\mathtt{S}}$, without loss of generality, we assume all other $\overline{\mathtt{Y}}$ has been substituted, then $\overline{\mathtt{T}}=\overline{\mathtt{S}}$. Then $\mathtt{C}<\overline{\mathtt{T}}>=\mathtt{C}<\overline{\mathtt{S}}>$, and by S-REFL, conclusion follows.

Case PM-CL-S: Follows from induction hypothesis.

□

LEMMA 23 SINGLE RANGE CONTAINMENT IS TRANSITIVE. *If* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$, $\Delta;[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_2\sqsubseteq_R R_3$, *then* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash R_1\sqsubseteq_R R_3$,

PROOF. By SB-R,
$R_1=(\mathtt{T}_1,\ <\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}>\overline{\mathtt{U}}\to\mathtt{U}_0)$ $R_2=(\mathtt{T}_2,\ <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>\overline{\mathtt{V}}\to\mathtt{V}_0)$
$\Delta\vdash\mathtt{T}_2<:\mathtt{T}_1$ $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}};[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash unify(\mathtt{U}_0:\overline{\mathtt{U}},\ \mathtt{V}_0:\overline{\mathtt{V}})$
$R_3=(\mathtt{T}_3,\ <\overline{\mathtt{Z}}\lhd\overline{\mathtt{O}}>\overline{\mathtt{V}}'\to\mathtt{V}'_0)$
$\Delta\vdash\mathtt{T}_3<:\mathtt{T}_2$ $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{O}};[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash unify(\mathtt{V}_0:\overline{\mathtt{V}},\ \mathtt{V}'_0:\overline{\mathtt{V}}')$
By S-TRANS, $\Delta\vdash\mathtt{T}_3<:\mathtt{T}_1$
By UNI,
$[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{U}_0:\overline{\mathtt{U}}=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{V}_0:\overline{\mathtt{V}}$ for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$
$[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}_0:\overline{\mathtt{V}}=[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$ for all $\mathtt{Z}_i\in\overline{\mathtt{Z}}$, $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{O}}\vdash\mathtt{Q}_i\prec:_{\overline{\mathtt{Z}}}\mathtt{Z}_i$
By construction of R_1 and R_2, no $\overline{\mathtt{Y}}$ appear in $\mathtt{U}_0:\overline{\mathtt{U}}$, no $\overline{\mathtt{Z}}$ appear in $\mathtt{V}_0:\overline{\mathtt{V}}$.
Thus, $\mathtt{U}_0:\overline{\mathtt{U}}=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{V}_0:\overline{\mathtt{V}}$, $\mathtt{V}_0:\overline{\mathtt{V}}=[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$
Then $\mathtt{U}_0:\overline{\mathtt{U}}=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$
Since no $\overline{\mathtt{Y}}$ appear in $\mathtt{V}'_0:\overline{\mathtt{V}}'$, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0:\overline{\mathtt{V}}'=[([\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{Q}})/\overline{\mathtt{Z}}]\mathtt{V}'_0:\overline{\mathtt{V}}'$
By inspecting PM-*, and by Lemma 5, it is clear that for all $\mathtt{Z}_i\in\overline{\mathtt{Z}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}},\overline{\mathtt{Z}}<:\overline{\mathtt{O}}\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{Q}_i\prec:_{\overline{\mathtt{Z}}}\mathtt{Z}_i$. The conclusion follows. □

LEMMA 24 METHOD TYPE LOOKUP TERMINATES.
Method type lookup $mtype(\mathtt{n},\ \mathtt{T})$ *for all* $\mathtt{T}$ *with a finite chain of reflective dependency either terminates with* $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{T})=\overline{\mathtt{U}}\to\mathtt{U}_0$, *or ends with none of the MT-* rules applicable.*

PROOF. The chain of reflective dependency is a sequence of types defined to be:

$$
\begin{array}{ll}
\mathit{refchain}(\texttt{Object}) & = \texttt{Object} \\
\mathit{refchain}(\texttt{X}) & = \texttt{X}{:}\mathit{refchain}(\mathit{bound}_{\Delta}(\texttt{X})) \\
\mathit{refchain}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) & = \texttt{C<}\overline{\texttt{T}}\texttt{>}{:}\mathit{refchain}(\texttt{T}_i){:}\mathit{refchain}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T}) \\
 & \text{where } \overline{\texttt{T}}{=}\texttt{T}_0,\ldots,\texttt{T}_n
\end{array}
$$

The chain is constructed so that if a reoccurrence of the same type, in any form of instantiation happens, the chain construction is terminated, and the chain is deemed not finite. Since there is a finite number of classes, the chain construction either terminates with a finite chain, or a reoccurrence as described above must happen.

We define the measure function to be:
$\mathit{measure}(\mathit{mtype}(\texttt{n},\ \texttt{T}))=\mathit{length}(\mathit{refchain}(\texttt{T}))$.

It is simple to see that with each recursive call, the measure must decrease:

Case MT-VAR-S:
$\mathit{measure}(\mathit{mtype}(\eta,\ \texttt{X}))=\mathit{length}(\mathit{refchain}(\texttt{X}))$
$\mathit{measure}(\mathit{mtype}(\eta,\ \mathit{bound}_{\Delta}(\texttt{X})))=\mathit{length}(\mathit{refchain}(\mathit{bound}_{\Delta}(\texttt{X})))$
Since $\mathit{length}(\mathit{refchain}(\texttt{X}))=1+\mathit{length}(\mathit{refchain}())\mathit{bound}_{\Delta}(\texttt{X})$, clearly measure decreases.

Case MT-SUPER-S,
$\mathit{measure}(\mathit{mtype}(\texttt{m},\ \texttt{C<}\overline{\texttt{T}}\texttt{>}))=\mathit{length}(\mathit{refchain}(\texttt{C<}\overline{\texttt{T}}\texttt{>}))$
$\mathit{measure}(\mathit{mtype}(\texttt{m},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}))=\mathit{length}(\mathit{refchain}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}))$
$\mathit{refchain}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})$ is embedded in $\mathit{refchain}(\texttt{C<}\overline{\texttt{T}}\texttt{>})$ by construction. Thus, the measure decreases.

Case MT-CLASS-R:
mtype is recursively invoked through *specialize* on one of the type parameters of $\texttt{C<}\overline{\texttt{T}}\texttt{>}$. By construction, again, $\mathit{refchain}(\texttt{T}_i)$ is embedded in $\mathit{refchain}(\texttt{C<}\overline{\texttt{T}}\texttt{>})$. Thus, again, measure decreases.

Case MT-SUPER-R: Similar to MT-CLASS-R and MT-SUPER-S.

Since the chains are finite, then the measure function cannot decrease infinitely. Thus, the recursion must terminate. □