Morphing with Filter Patterns: Bringing Discipline to Meta-Programming

Shan Shan Huang
College of Computing
Georgia Institute of Technology
ssh@cc.gatech.edu

Yannis Smaragdakis
Department of Computer and Information Science
University of Oregon
yannis@cs.uoregon.edu

Abstract

Morphing is a language feature for specifying classes whose members are produced by statically iterating over members of other classes. Morphing adds significant expressiveness to a programming language. A similar effect is otherwise achievable only with techniques such as meta-programming, or aspect-oriented programming. In contrast to such techniques, morphing is a natural extension of traditional genericity, and is modularly type safe, allowing separate reasoning for each class, despite not knowing its exact members.

In this paper, we present a significant enhancement to the idea of morphing. In addition to patterns, which specify members to iterate over, we enable *filter patterns*: ways to state existential conditions (depending on the presence of other members) that enable or block the applicability of a pattern. Filter patterns dramatically augment the expressiveness of morphing. Many real-world examples require filter patterns in order to be implemented type-safely. We demonstrate the power of filter patterns with concrete applications, and present a type-checking algorithm. We offer a decidability proof for the type-checking of morphing with filter patterns and argue that our language represents a sweet spot in the spectrum of expressiveness vs. reasoning ability.

1. Introduction

The design of programming languages largely consists of balancing the contrasting demands of expressiveness and reasoning power. We recently proposed the idea of *morphing* [11] to target a long-time need in programming: that of writing generic code separately and having it be applied appropriately to multiple sites (e.g., methods or fields) of an application. Mechanisms such as meta-object protocols, aspect-oriented programming, or program transformation are often employed in such circumstances. Nevertheless, all of them suffer from lack of modular reasoning, particularly in terms of type safety: It is not possible to establish the well-formedness of the generic code independently from the type parameters it is being instantiated with. Morphing supports this common need for expressiveness without sacrificing modular reasoning and with a smooth extension of traditional genericity, instead of introducing concepts such as aspects or transforms.

Our reference morphing implementation is a Java-based language called MJ. For a realistic example of morphing with MJ, consider class `Collections` in the Java Collections Framework (JCF)—the standard Java data structures library. `Collections` defines over 30 nested classes whose purpose is to wrap existing JCF classes and interfaces with extra functionality. For instance, one such class is `Collections.SynchronizedList`, which corresponds to the `List` interface. The code for this class has a lot of redundancy. For each method of the `List` interface, `SynchronizedList` has a method with identical signature, which synchronizes on a mutex and then delegates to the original `List` method. The redundancy extends to many other classes (`SynchronizedSet`, `SynchronizedMap`, `SynchronizedSortedSet`, and more). Yet, all these classes and all the similarly structured methods inside them can be replaced with a single instance of morphing, as in class `MakeSynchronized` below:[1]

```
public class MakeSynchronized<X> {
  X x;
  public MakeSynchronized(X x) { this.x = x; }

  //For each non-void method in X, declare the following:
  <R,A*>[m] for(public R m(A) : X.methods)
  public synchronized R m (A a) { return x.m(a); }

  // Similarly for each void-returning method in X
  <A*>[m] for(public void m(A) : X.methods)
  public synchronized void m(A a) { x.m(a); }
}
```

The body of `MakeSynchronized` is defined by “reflective” static iteration (using the two `for` statements) over all methods of `X` that match the two patterns “`public R m(A)`” and “`public void m(A)`”. This static-`for` loop is the main language feature of MJ. `R`, `A`, and `m` are pattern variables. `R` and `A` match types, while `m` matches method names. Additionally, the `*` symbol following the declaration of `A` indicates that `A` matches any number of types (including zero). That is, the first pattern matches all `public` methods that have a return type, while the second matches all `public` methods that return nothing. The pattern variables are used in method declarations: for each method of the type parameter `X`, `MakeSynchronized` declares a method with the same name and type signature. (This does not have to be the case, as shown in later examples.) Thus, the exact methods of class `MakeSynchronized` are not determined until it is type-instantiated. We can produce the required synchronized classes of the JCF by just using the type-instantiations `MakeSynchronized<Map>`, `MakeSynchronized<List>`, `MakeSynchronized<Set>`, etc.

[1] The code is slightly simplified in ways that do not impact our discussion—e.g., the full version allows the user to specify his/her own mutex, as well as a superclass or interfaces.

The full version of the above MJ class consists of less than 50 lines of code, replacing more than 600 lines of code in the JCF. Similar simplifications can be obtained for other nested classes in `Collections`, which account in total for some 2000 lines of code in the original JCF implementation: `UnmodifiableSet`, `UnmodifiableList`, `UnmodifiableMap`, etc. are replaced by a single morphed class, and the same is done for `CheckedSet`, `CheckedList`, `CheckedMap`, etc. Note that morphing allows us to eliminate redundancy both within the same class (e.g., the 15-some methods of `SynchronizedMap` that all just acquire a mutex and delegate to the corresponding method of a `Map` object) and across classes (the same needs to be done for `Set`, `List`, `SortedSet`, and more). Additionally, a morphed class adapts effortlessly to changing interfaces: any changes to the method signatures of `List` are reflected automatically in `MakeSynchronized<List>`.

The modular type safety guarantee of MJ means that highly generic classes like `MakeSynchronized` can be type-checked separately from their type-instantiations. If the definition of `MakeSynchronized<X>` passes type-checking, then *any* type-instantiation `MakeSynchronized<C>` represents a valid class, regardless of the contents of type `C`. (Although assumptions for `C` can be stated in the definition, as constraints on `X`.) Modular type safety is a crucial property, and is often one that separates high-level language mechanisms from low-level, macro-based approaches. In practice, it means that a developer can freely write generic code with the same guarantees as straightforward methods. There is no chance that a type error (e.g., an accidental use of a too-general type) will remain hidden in most uses of a generic class, only to be exposed much later when an insidious type-instantiation comes along. For this reason, we view morphing as an elevation of reflective meta-programming features to disciplined language constructs.

Despite its power, morphing with just static iteration and pattern matching is not sufficient to express some common tasks with modular type safety. The essence of the type guarantees of morphing hinges on reasoning about *subsumption* and *disjointness* of ranges of declarations described by static-`for` patterns. Subsumption is necessary for establishing that every reference has a corresponding definition, and disjointness for establishing that no two definitions conflict, rendering the resulting class ambiguous. Establishing disjointness is the hardest part of type checking morphing features, and this capability is commonly omitted in other languages that attempt to offer type-safe reflection (e.g., CTR [8]). An elegant way to make disjointness provable in a much wider variety of cases is with the introduction of *filter patterns*: patterns that express enabling conditions during static iteration. For instance, consider using morphing to create a subclass of a given class decorated with accessor and mutator methods (`get`/`set` methods) for each field:

```
public class AddGetSet<class X> extends X {
  <T>[f] for(T f : X.fields) {|
    public T get#f () { return f; }
    public void set#f (T nf) { f = nf; }
  |}
}
```

This example uses more MJ language features: We iterate over fields instead of methods. There are multiple declarations under a single pattern, using `{|`...`|}` as delimiters. The type parameter `X` is required to be a class (not an interface). Furthermore, class `AddGetSet` is a mixin: it extends its type parameter. Finally, we use the `#` operator to add a constant prefix ("`get`" or "`set`") to a name variable (the name of field `f`).

The above code does not pass MJ type checking, though, and for good reason. The type parameter `X` may already contain a conflicting method—e.g., a class `Foo` can contain a field "`int up`", but also a method "`int setup(int i)`". In this case, `AddGetSet<Foo>` is ill-defined: it contains a method "`void setup(int)`", which does not correctly override method "`int setup(int)`" in the superclass.

To express this functionality correctly, we need a filter pattern to guarantee that no method in the type parameter conflicts with the ones being introduced. The resulting correct class becomes:

```
public class AddGetSet2<class X> extends X {
  <T>[f] for(T f : X.fields ; no get#f() : X.methods)
  public T get#f () { return f; }

  <T>[f] for(T f : X.fields ; no set#f(T) : X.methods)
  public void set#f (T nf) { f = nf; }
}
```

The filter patterns are introduced under the `for` statement with the `no` keyword, and look just like regular (*primary*) patterns. In this case, the filter patterns ensure that `X` contains no methods conflicting with the `get`/`set` methods introduced. Thus, the filter patterns are *negative* patterns in this case. A negative filter pattern is *not* just a negated predicate over the methods that the primary pattern iterates over. Instead, the filter pattern introduces an extra existential condition: this condition is satisfied if there *exists no member* matching the filter pattern. Thus, a good way to read, e.g., the first of the above static-`for` loops is: "for all fields `f` in `X`, for which there exists no method `get#f` (i.e., a prefix `get` and the name of the field) in `X`...". Also, filter patterns are not just extra primary patterns: the primary pattern holds a special role, since it binds all the pattern variables that can be used in the morphed code.

Filter patterns have significant practical impact. Most of the real-world examples we discuss are only possible in a modularly type-safe manner because of filter patterns. Thus, this paper makes several concrete contributions:

- We introduce the concept of filter patterns, which drastically enhances the expressiveness of morphing. A limited form of filter patterns was mentioned as future work in our earlier publication on morphing [11]: we had only envisioned negative filter patterns, but positive filter patterns turn out to be just as useful.
- We formalize our type checking algorithm with filter patterns and prove it sound. We also prove the decidability of type-checking with filter patterns. Previous morphing type checking (even without filter patterns) did not have a decidability proof.
- We show large-scale examples of the real-world applicability of morphing (one of which is the rewriting of the `Collections` class in the JCF, as already discussed above). This establishes morphing as a key technique with significant benefit for common programming tasks. Earlier morphing work had not demonstrated its benefits on real-world code or well-known problems from the literature.
- The current formulation of morphing represents an interesting balance between expressiveness and reasoning power. General static iteration cannot have a type checking algorithm that is both complete (i.e., does not conservatively reject programs) and decidable. Yet our type checking algorithm is complete under broad assumptions, and easily supports in practice all the useful examples of morphing we have encountered.

2. MJ Type-Checking Background

We next discuss the main type-checking insights of MJ with only primary patterns. We present the ideas in a form that can be easily adapted to carry over to MJ with filter patterns. Due to limited space, we jump directly to the type checking issues without a further introduction of the original MJ language constructs—these were already shown in the examples above and the interested reader can consult our previous publication [11] for more detail.

The complexity of type-checking MJ programs compared to plain Java programs is due to having reflective members with unknown names and types. In a block of code under a static-`for` statement, a single identifier or type name can refer to many possible program elements, with the only constraint being that they match the pattern used for static iteration. We call the set of these elements the *reflective range*, or just *range*, of the identifier or type variable.

The job of MJ'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—this is the meaning of modular type safety. The key to establishing type safety is the ability to statically check subsumption and disjointness for reflective ranges.

2.1 Internally Well-Defined Ranges

A simple property to establish is that declarations introduced by the same static-`for` loop do not conflict. Consider the following:

```
class CopyMethods<X> {
  <R,A*>[m] for( R m (A) : X.methods )
  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, 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 (the base case being `Object`), 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.

It is important to make sure 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.

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. E.g.:

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

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 )
  A m ( R a ) { ... }
}
```

In this example, the uniquely identifying part of `X`'s method is no longer the uniquely identifying part of `IllegalChange<X>`'s method: the argument type of `X`'s method is no longer part of the argument type of `IllegalChange<X>`'s method. An error would occur if we instantiate `IllegalChange` with a class with two one-argument methods with the same name and return type.Therefore, the `IllegalChange` definition is rejected by MJ's type system.

2.2 Range Subsumption

The main challenge of modular type checking for a morphing language is to ensure the validity of references: a referenced entity should have a definition, and its use should be well-typed.

An easy case is that of references to reflective ranges whose exact contents are statically known. For instance, we can have a static-`for` loop over "`C.methods`", where `C` is a known class. This case is uninteresting from a typing perspective, however: the static-`for` is a mere syntactic convenience, and the reasoning is the same as if every iteration of the static-`for` was inlined statically. We generally ignore this case in our typing discussion.

Generally, to establish reference validity, we need to determine *range subsumption*: do all entities in the reference range have a corresponding entity in the declaration range? An easy case is that of an unbounded reflective range (i.e., a range over an unknown type) that refers to entities in a statically known range. This is never safe, as the unknown range is unbounded and can contain more entities. For instance, consider the example:

```
class InvalidReference<X> {
  Foo f;  ... // code to set f field
  [n] for( void n (int) : X.methods )
  void n (int a) { f.n(a); }
}
class Foo {
  void foo(int a) { ... }
}
```

The reference `f.n(a)` is unsafe, since there are certainly type-instantiations of `InvalidReference` for which name variable `n` will range over more method names than just `foo`, and, thus, the call `f.n(a)` will cause a compilation error.

Another easy case is that of a reflective range that refers to the entities it iterates over. For instance:

```
class EasyReflection<X> {
  X x;  ... // code to set x field
  [n] for( void n (int) : X.methods )
  void n (int i) { x.n(i); }
}
```

Although the range of methods that `n` can refer to is unbounded, they are guaranteed to be in one-to-one correspondence with the methods declared for `X`. Thus, the declaration range subsumes the reference range and reference `x.n(i)` is clearly valid, as long as the usual Java type compatibility checks hold—e.g., `i` is an `int`, which is what `n` expects.

The interesting and complex case is that of references under a reflective iterator to methods and fields declared under *different* reflective iterators. Consider the following example:

```
class Reference<X> {
  Declaration<X> dx;  ... // code to set dx field
  <A*>[n] for( String n (A) : X.methods )
  void n (A a) { dx.n(a); }
}
class Declaration<Y> {
  <R extends Object,B*>[m] for( R m (B) : Y.methods )
  void m (B b) { ... }
}
```

To check the validity of method invocation `dx.n(a)`, we need to determine that the range of `n` in `Reference<X>` is subsumed by the declaration range of methods in `Declaration<X>`. We observe that the two ranges are over methods of the same type, `X`. The range of `n` in `Reference<X>` consists of the names of methods in `X` that return a `String` type. The method names in `Declaration<X>` are the names of methods in `X` that return a non-primitive type. Thus, the latter range subsumes the former. This guarantees that `Declaration<X>` does have a method matching each `n`.

Subsumption of ranges in the MJ type system is checked by unification of reflective patterns. The signature (name and types) of the declaration pattern should unify with the signature of the reference pattern, using the pattern type and name variables of the former as unification variables, and treating all types and names in the latter as constants (i.e., one-way unification). This ensures that the declaration range is more general than the reference range, hence there exists a corresponding declaration for every entity in the reference range. For instance, the above example class `Reference` is safe because it is possible to unify range "`R m (B)`" with range "`String n (A)`", using only `m`, `R`, and `B` as variables.

To guarantee that the call `dx.n(a)` is well-typed, we also need to confirm that the actual argument type is compatible with the formal argument type for methods in `Declaration<X>`. This is simple, because, after unification, type variables `A` and `B` are equated. Since the type of argument `a` is just `A`, while the expected type (for methods in `Declaration<X>`) is `B`, the call `dx.n(a)` is valid.

Our formal type system in Section 5 expresses precisely this idea of testing range subsumption using unification. Using unification for range comparisons (both subsumption and disjointness) is the main idea of the MJ type system. There is, however, a subtlety that we brush over in our informal discussion. (We discuss this in Section 5.) Our unification test is followed by checking of type bounds (i.e., the known supertypes of type variables) for compatibility. In the above example, the two signatures are unified by mapping `R` to `String`. `R` is bounded by `Object`. Since `String` is a subtype of `Object`, this is a valid unification mapping.

2.3 Range Disjointness

We have discussed how to determine declaration uniqueness within one reflective block. Another requirement, however, is that the declarations in a reflective block do not conflict with declarations outside the reflective block. This is a matter of showing *range disjointness* among all ranges of declarations that may conflict.

Just as in the case of range subsumption, an easy disjointness test involves a statically known range and an unknown one. In this case, we cannot guarantee disjointness, unless static prefixes are used. 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 StaticName<X> {
  int foo () { ... }

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

A static prefix on the reflectively declared method names (e.g., `bar#m` instead of just `m`) can guarantee that all declarations produced have names distinct from `foo` (unless, of course, the static prefix is also a prefix of the identifier in an existing declaration).

Disjointness requirements do not only apply among ranges of declarations in the same class. A declaration may also conflict with another declaration in a supertype. Proper method overriding implies 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. 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 [4]. Since the supertype could potentially be any type, we have no way of knowing its members. 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 X> extends X {  void foo () {}  }
```

Thus, any legal type extending its type parameter can contain *no* members other than reflective iterators over its supertype that declare properly overriding versions for (some subset of) the supertype's methods. (Later we show how filter patterns can be used to relax this restriction by ensuring that no conflicts arise.)

The full complexity of testing range disjointness becomes apparent when we have multiple reflective ranges, and type variables are used to form complex types. Consider the following example:

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

  <P>[n] for ( P n (X) : X.methods )
  P n (List<X> a) { ... }
}
```

In the two reflective blocks of `ManipulationError<X>`, different manipulations are applied to the uniquely identifying parts—in the first block, no manipulation is applied, while in the second block, the argument type is changed to `List<X>` from `X`. Even though the two reflective blocks iterate over distinct sets of methods, they do *not* have disjoint declaration ranges. One instantiation that would cause a static error is the following:

```
class Conflicting {
  int m1 ( List<Conflicting> a ) { ... }
  int m1 ( Conflicting a ) { ... }
}
```

`ManipulationError<Conflicting>` would contain two methods named `m1`, both taking argument `List<Conflicting>`.

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 instance, our earlier example can be rewritten correctly as:

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

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

Disjointness of ranges in the MJ type system is again checked by unification, but this time of the signatures of members being declared (not the iteration patterns). If the uniquely identifying parts of the two signatures (i.e., their names and argument types) unify, then the ranges are *not* disjoint. In our `ManipulationError` example above, the two signatures are "`R m (List<X> a)`" and "`P n (List<X> a)`", with type variables `R` and `P`, and name variables `m` and `n`. The two signatures clearly unify, hence the reflective ranges are not disjoint.[2]

An important point is that the test for range disjointness is a *two-way* unification, unlike the one-way unification used for range subsumption. That is, the unification test for range disjointness treats

[2] Note that the pattern used for reflective iteration has no bearing on the disjointness check. E.g., in class `ManipulationError` it makes no difference that the patterns "`R m(List<X>)`" and "`P n(X)`" refer to distinct methods: As long as the variables (in this case `P`, `R`, `m`, `n`) get their values from an unknown type (in this case `X`), we have few means to guarantee that they do not overlap, as our patterns give no way to restrict the structure of what is matched by a variable (only the structure of what is *around* the variable).

the pattern type and name variables in both ranges as unification variables. For instance, consider the following example:

```
class WhyTwoWay<X> {
  <A1,R1> for ( R1 foo (A1) : X.methods )
  void foo (A1 a, List<R1> r) { ... }

  <A2,R2> for ( R2 foo (A2) : X.methods )
  void foo (List<A2> a, R2 r) { ... }
}
```

This morphed class is not type safe. The two ranges have a possible overlap. Although neither is more general than the other, the signatures "`void foo(A1, List<R1>)`" and "`void foo(List<A2>, R2)`" (with variables `A1`,`R1`,`A2`,`R2`) do unify. Concretely, the method `foo` is not well-defined in class `WhyTwoWay<C>` if class `C` contains two methods with signatures "`String foo(List<String>)`" and "`List<String> foo(String)`". Testing range disjointness via two-way unification is expressed precisely in our formalism (Section 5).

3. Filter Patterns

We next enhance MJ to allow filter patterns, in addition to regular patterns. Filter patterns are interesting because they can represent *existential* conditions on the type parameter, in addition to conditions on the current method or field being matched. Filter patterns play the role of a powerful conditional statement. The added expressiveness enables many practical programming tasks.

3.1 Filter Pattern Syntax

Filter patterns have the same form as the primary pattern of an MJ static iterator, but are preceded by the keywords "`some`" (for *positive* filter patterns) or "`no`" (for *negative* filter patterns). There can be only a single filter pattern per static iterator, and it has to follow the primary pattern, separated by a semicolon. (Multiple filter patterns for a single primary pattern can be easily simulated by using intermediate types.)

A negative filter pattern states that there should exist no members in the range designated by the pattern. The pattern elements themselves are determined by the current member being iterated over (by the primary pattern). We saw in the Introduction an example: "`<T>[f] for(T f : X.fields ; no get#f() : X.methods)`". This static iterator ensures that for each field `f` in its range, class `X` has no method matching the pattern "`get#f()`". (The missing return type in the filter pattern is an MJ shorthand for matching both void and non-void return types.)

Positive filter patterns ensure that there exists *some* member in the range designated by the pattern. For example,

```
public class DoBoth<X,Y> {
  <A*>[m] for(  static void m(A) : X.methods ;
           some static void m(A) : Y.methods )
  public static void m(A args) {
    X.m(args);
    Y.m(args);
  }
}
```

`DoBoth<X,Y>` finds all static methods of `X` that have a counterpart with identical signature in `Y`. For each of those, `DoBoth<X,Y>` defines a method with the same signature and a body that calls first the corresponding method of `X`, and then that of `Y`.

A filter pattern introduces a second iteration, nested inside the static iteration of the primary pattern. Therefore, a similar effect could be achieved by allowing nested static-`for` loops (and negation). However, filter patterns are only used to return a true/false decision. For instance, in class `DoBoth` above, the filter pattern iterates over all the methods in the second type parameter, `Y`, but the iteration only serves to verify whether a matching method exists or not, and not to produce different code for each matching method in `Y`. This is also reflected in a syntactic restriction: filter patterns can introduce new pattern (type or name) variables, but these variables cannot be used in the code being introduced by the static iterator. For instance, a filter pattern usage from an MJ example application (Section 4) is

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

This static iteration has no primary pattern and uses the filter pattern only to ensure that no method in class `X` conflicts with the one being introduced. Although the filter pattern introduces pattern type variable `R`, this cannot be used in the declaration of `restore`.

The reader may wonder why we chose filter patterns instead of allowing arbitrary nested iteration. Our goal was to define morphing language features to be powerful enough for practical use, yet also constrained enough for automatic reasoning. Arbitrary iteration makes range subsumption and disjointness tests undecidable. We discuss the issues of completeness and decidability in Section 6.

3.2 Filter Pattern Type Checking

Type checking of a morphing language is based on range subsumption and disjointness tests. In the presence of filter patterns, we can express range subsumption and disjointness tests in terms of similar tests for single-pattern ranges. We use the notation $\langle P, +F\rangle$ and $\langle P, -F\rangle$ to express a range with primary pattern P, and positive or negative filter pattern F. Our subsumption rules are then as follows:

- $\langle P_1, +F_1\rangle$ subsumes $\langle P_2, +F_2\rangle$ if P_1 subsumes P_2, and F_1 subsumes F_2.
- $\langle P_1, -F_1\rangle$ subsumes $\langle P_2, -F_2\rangle$ if P_1 subsumes P_2, and F_2 subsumes F_1.

For testing disjointness of declarations, there are three patterns at play: the primary pattern we are iterating over, the filter pattern, and the pattern representing the signature of the member being declared. We extend our notation with a declaration pattern G: we describe a range as $\langle P, +F, G\rangle$, $\langle P, -F, G\rangle$, or $\langle P, ?F, G\rangle$, where $?F$ stands for either a positive or negative filter pattern. Our disjointness rules are:

- $\langle P_1, ?F_1, G_1\rangle$ is disjoint from $\langle P_2, ?F_2, G_2\rangle$ if G_1 is disjoint from G_2.
- $\langle P_1, ?F_1, G_1\rangle$ is disjoint from $\langle P_2, -F_2, G_2\rangle$ if F_2 subsumes P_1.
- $\langle P_1, +F_1, G_1\rangle$ is disjoint from $\langle P_2, -F_2, G_2\rangle$ if F_2 subsumes F_1.

Note that without filter patterns only the first rule applies, which reflects exactly what we discussed earlier (Footnote 2): for disjointness tests without filter patterns, the patterns that are compared are G_1 and G_2, not P_1 and P_2.

We showed all the above rules in simplified form for conciseness and ease of exposition. This elides some important details:

- For the rules to work in boundary cases, an empty primary pattern can be thought of as a primary pattern that subsumes everything. An empty filter pattern can be thought of as a positive filter pattern that subsumes everything, or a negative filter pattern subsumed by everything.
- When checking subsumption by a filter pattern (e.g., "F_1 subsumes F_2") the check is done after applying the substitution of pattern variables dictated by the unifier of primary patterns P_1 and P_2 (for subsumption checks) or of declared member patterns G_1 and G_2 (for disjointness checks). This is possible because, in these rules, F_1 and F_2 need to be consulted only when

P_1 subsumes P_2 (i.e., there is one-way unification) or when G_1 is *not* disjoint from G_2 (i.e., there is a two-way unification).

- The above disjointness rules do not fully capture Java's overloading restrictions (i.e., that two overloaded methods cannot only differ in their return type). We considered presenting the full set of rules, but found the extra complexity a distraction from the main concepts of this paper.[3]

We next illustrate these rules with informal examples.

3.2.1 Range Subsumption

Consider two ranges with positive filter patterns. We can always refer to a member declared under a more general primary pattern and a more general filter pattern. For instance, a refinement of our earlier `DoBoth` example is the following:

```
public class DoBoth2<X,Y> {
  <A*>[m] for(   static void m (A) : X.methods ;
            some static void m (A) : Y.methods)
  public static void my#m (A args) {
    X.m(args);
    Y.m(args);
  }
  public static void callAllIntMeths(int i) {
    [n] for(   static void n (int) : X.methods ;
          some static void n (int) : Y.methods)
    {|  my#n(i);  |}
  }
}
```

`DoBoth2<X,Y>` declares a method for each void-returning method with identical signature (including name) in both `X` and `Y`. It also declares a method `callAllIntMeths` whose body calls in sequence all of the newly declared methods that take a single integer argument. The call is safe, since each method designated by `my#n` is guaranteed to exist: the declaration range subsumes (i.e., is more general than) the reference range.

For negative filter patterns, the obligation is the opposite: for a range to be more general, its negative pattern needs to be more restrictive. Consider the following example:

```
public class PosAndNeg<X,class Y> extends Y {
  X x; // ... code to set field x;

  <A*,R>[m] for(void m (A) : X.methods ;
                no R m (A) : Y.methods)
  public void m(A arg) { x.m(arg); }

  <B*,P>[n] for (void n (int) : X.methods ;
                 no P n (B)   : Y.methods)
  public void zero#n() { n(0); }
}
```

`PosAndNeg<X,Y>` defines a method `m` for each void-returning method of `X`, for which there is no conflicting method in `Y`. (This is necessary for well-formedness, since `Y` is a supertype of `PosAndNeg<X,Y>`.) The second static iteration goes over those of the earlier-declared methods that only take an integer argument, and for each one declares a method that calls it with `0`. The invocation of `n` in "`n(0)`" is well-typed because the primary pattern of the declaration of `m` (i.e., "`void m(A) : X.methods`") subsumes the primary pattern of the invocation (i.e., "`void n(int): X.methods`"), under a unifier mapping `m` to `n`, `A` to `int`. The negative filter pattern of the declaration (i.e., "`R m(int) : Y.methods`", after the substitution dictated by the primary patterns' unifier) is subsumed by the negative filter pattern of the use (i.e., "`P n(B) : Y.methods`").

3.2.2 Range Disjointness

One of the most valuable uses of filter patterns is for proving range disjointness. Note that filter patterns cannot affect the *structure* of the members being declared in a static iteration, but only whether the members are there at all. Therefore, if two ranges have disjoint patterns for declaring members, then they are disjoint regardless of their primary or filter patterns.

A more interesting case concerns the interaction of a negative filter pattern and a primary pattern. In the following example, the generated declarations may conflict according to their signatures (i.e., can have overlapping members) but the negative filter pattern is sufficient to show that the elements causing the overlap cannot satisfy the filter condition (i.e., are not part of the second static iteration range). (This is the most complex informal example in the paper. Readers who follow the discussion below have grasped the essence of our type-checking.)

```
public class UnionOfStatic<X,Y> {
  <A*>[m] for( static void m (A) : X.methods)
  public static void m(A args) { X.m(args); }

  <B*>[n] for( static void n (B)      : Y.methods;
            no static void n (int, B) : X.methods)
  public static void n(int count, B args) {
    for (int i = 0; i < count; i++) Y.n(args);
  }
}
```

`UnionOfStatic<X,Y>` first copies all static, void-returning methods of `X`, making forwarding methods for them. A second reflective iterator goes over each static, void-returning method of `Y`, and declares a corresponding method with an additional integer argument. The second range has a negative filter pattern, which excludes all possibly conflicting methods that come from the first range. This is necessary, or the two ranges would not be disjoint. We can see in detail how the disjointness follows from our earlier rules. The two patterns of declared methods are "`static void m(A)`" and "`static void n(int, B)`". These are not disjoint: they unify with a substitution of "`int,B`" for "`A`" and "`n`" for "`m`" (or vice versa). (The former is possible since type variable `A` is declared as `<A*>`, i.e., can represent a sequence of types.) Applying this substitution to the primary pattern of the first range yields "`static void n(int,B)`". After the substitution, the (negated) filter pattern of the second range also becomes "`static void n(int,B)`" (both patterns over `X.methods`). The negated filter pattern subsumes the other range's primary pattern. Thus, an element satisfying the negated filter cannot be part of the first range, establishing disjointness.

A very similar argument shows that disjointness can be established if a negative filter pattern subsumes a positive filter pattern of the other range, which corresponds to the last of our above rules.

4. Real-World Examples

We saw in the Introduction how the `Collections` class of JCF can be rewritten very concisely using MJ. We next examine two more real-world applications of morphing. The first is an example of meta-programming tools used in prior literature [9, 16]. The second implements support for software transactional memory and was used as "the quintessential example" in Fähndrich, Carbin, and Larus's Compile-Time Reflection (CTR) system [8]. CTR offers no guarantee of modular safety with respect to conflicting definitions, however. Neither of these examples can be written type-safely without filter patterns.

[3] For reference, the complete set of rules for overloading has two versions of each of the three disjointness rules. The three rules given above apply when the uniquely identifying parts of G_1 and G_2 come from iterating over members of the same type (in which case we know no conflict exists). Three slightly different rules are needed to capture the general case: the tests of disjointness or subsumption of single ranges need to be over the uniquely identifying parts of a signature, while the non-uniquely identifying part (return type) of the signature needs to be free in negative filter patterns.

Java ensures that a class cannot be declared to "implement" an interface unless it provides implementations for all of the interface's methods. This often results in very tedious code. For instance, it is common in code dealing with the Swing graphics library to implement an event-listener interface, yet provide empty implementations for most of the interface methods because the application does not care about the corresponding events. In response to this need, there have been mechanisms proposed [9, 16] that allow the programmer to specify that he/she wants just a default implementation for all members of an interface that are not already implemented by a class. These past solutions introduced new keywords (or Java annotations), and were implemented using program transformation. In contrast, the MJ solution is expressed using MJ's extension of Java generics, and offers modular type safety, instead of unsafe, low-level program transformation.

Below is a slightly simplified version of the MJ code. (For conciseness, we elide the static iterators dealing with void-returning methods. These roughly double the code.)

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

  // for all methods in I, if the same method does
  // not appear in X, provide default implementation.
  <R,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); }
}
```

Class `MakeImplement<X,I>` imitates the public interface of type `X` 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) are ignored. The above code demonstrates the power of filter patterns, both in terms of expressiveness, and in 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, the three ranges are designed so they are guaranteed disjoint, and the disjointness is crucially based on the use of filter patterns.

4.1 Making Code Transactional

Our second example of filter patterns reproduces an application from Fähndrich, Carbin, and Larus's Compile-Time Reflection (CTR) system [8]. The application is a re-implementation of Herlihy's SXM (a C#-based Software Transactional Memory system). The original implementation (by Herlihy) used run-time reflection and was about 500 LOC long. The re-implementation with CTR is an order of magnitude smaller (about 60 LOC) and the same conciseness is maintained in our morphing implementation (about 65 LOC + 10 LOC for emulating C# "properties" with explicit get/set methods).

This application was "the quintessential example" of CTR usage [8], since it has practical value and a standard reflective structure. The input class has its fields annotated with `@atomic` Java annotations. This means that reads and writes to these fields should be done through transactional operations (calling a low-level transaction library). The morphing class iterates over all `@atomic` fields of the input class and creates get/set methods that wrap field accesses with transactional operations. The system also adds shadow fields and backup/restore methods to facilitate object rollback on aborted transactions. The resulting class implements `IRecoverable`, which indicates it supports `backup()` and `restore()` methods. The MJ code is shown in its entirety in Figure 4.1. It uses interfaces `TransactionManager` and `Transaction` to communicate with the low-level transaction library.

The CTR version of the code is more client-friendly through the use of C# properties: C# allows using field access syntax that instead redirects through get/set methods. In the MJ version, client code needs to use get/set methods explicitly. This difference is orthogonal to our discussion, however, and depends on the underlying programming language only.

The advantage of MJ over the CTR implementation is full modular type-safety. CTR detects undefined variable errors, but offers no guarantee of modular safety with respect to conflicting (e.g., duplicate or improperly overriding) definitions. To achieve this benefit, the MJ version is heavily based on filter patterns. For example, consider the introduction of a backup field, for each original field annotated `@atomic`:

```
<F>[f]for (@atomic F f : X.fields;
        no F backup#f : X.fields )
F backup#f;
```

As can be seen in Figure 4.1, similar filter patterns are used in all other definitions (restore fields, backup and restore methods, get/set methods, etc.).

5. Formalization

We present a formalization, FMJ, which is based on FGJ [12], with differences (other than the simple addition of our extra environment, Λ) highlighted in gray.

5.1 Syntax

The syntax of FMJ is presented in Figure 2. 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; `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: `#` denotes a variable method name; `n` denotes either variable or non-variable names; o denotes a filter condition operator (either + or -) for the keywords `some` or `no`, respectively.

We use the shorthand $\overline{\mathtt{T}}$ for a sequence of types $\mathtt{T}_0, \mathtt{T}_1, \ldots, \mathtt{T}_n$, and $\overline{\mathtt{x}}$ for a sequence of unique variables $\mathtt{x}_0, \mathtt{x}_1, \ldots, \mathtt{x}_n$. We use : for sequence concatenation, e.g. $\overline{\mathtt{S}}$:$\overline{\mathtt{T}}$ is a sequence that begins with $\overline{\mathtt{S}}$, followed by $\overline{\mathtt{T}}$. We use $\in$ to mean "is a member of a sequence" (in addition to set membership). We use $\ldots$ for values of no particular significance to a rule. $\triangleleft$ and $\uparrow$ 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 MJ features that are representative of our approach. One simplification is that we do not allow a filter pattern to 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.

```
class MakeTransactional<C> implements IRecoverable {
  // for each field f annotated with @atomic,
  // generate current#f
  <F>[f]for (@atomic F f : C.fields ;
             no F current#f : C.fields )
  F current#f;

  // for each field f annotated with @atomic,
  // generate backup#f
  <F>[f]for (@atomic F f : C.fields ;
             no F backup#f : C.fields )
  F backup#f;

  // for each field f annotated with @atomic,
  // generate a backup and restore method
  <F,R>[f]for (@atomic F f : C.fields ;
               no R backup#f() : C.methods )
  void backup#f () { backup#f = current#f; }

  <F,R>[f]for (@atomic F f : C.fields ;
               no R restore#f() : C.methods )
  void restore#f () { current#f = backup#f; }

  // if C doesn't already have restore(), generate it.
  <R>for ( ; no public R restore() : C.methods)
  public void restore() {
    <F>[f] for(@atomic F f: C.fields)
    restore#f();
  }

  // if C doesn't already have backup(), generate it.
  for ( ; no public void backup() : C.methods )
  public void backup() {
    <F>[f] for(@atomic F f: C.fields)
    backup#f();
  }

  // for each field f annotated with @atomic,
  // generate getter/setter
  <F>[f]for (@atomic F f : C.fields ;
             no get#f() : C.methods )
  public F get#f() {
    Transaction me =
      TransactionManager.getCurrentTransaction();
    Transaction conflict = null;

    while(true) {
      synchronized(this) {
        conflict =
          TransactionManager.openForRead(this, me);
        if ( conflict == null ) return current#f;
      }
      TransactionManager.resolveConflict(me, conflict);
    }
  }

  <F>[f]for (@atomic F f : C.fields ;
             no set#f() : C.methods )
  public void set#f(F val) {
    Transaction me =
      TransactionManager.getCurrentTransaction();
    Transaction conflict = null;

    while (true) {
      synchronized(this) {
        conflict =
          TransactionManager.openForWrite(this, me);
        if ( confilct == null ) current#f = val;
      }
      TransactionManager.resolveConflict(me, conflict);
    }
  }
}
```

Figure 1. Making a class "transactional".

T	::=	$X \mid N$
N	::=	$C<\overline{T}>$
CL	::=	$\text{class } C<\overline{X}\lhd\overline{N}>\lhd\ N\ \{\overline{T}\ \overline{f};\ \overline{M}\}$
	$\mid$	$\text{class } C<\overline{X}\lhd\overline{N}>\lhd\ T\ \{\overline{T}\ \overline{f};\ \overline{\mathfrak{M}}\}$
M	::=	$T\ m\ (\overline{T}\ \overline{x})\ \{\uparrow e;\}$
$\mathfrak{M}$	::=	$<\overline{Y}\lhd\overline{P}>\text{for}(\mathbb{M}_p;o\mathbb{M}_f)\ U\ \#\ (\overline{U}\ \overline{x})\ \{\uparrow e;\}$
o	::=	$+ \mid -$
$\mathbb{M}$	::=	$V\ \#\ (\overline{V}):X.\text{methods}$
e	::=	$x \mid e.f \mid e.\ n\ (\overline{e}) \mid \text{new } C<\overline{T}>(\overline{e})$
n	::=	$m \mid \#$

Figure 2. Syntax

Lastly, static name prefixes, casting expressions and polymorphic methods are not formalized.

An important point is that 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 filter patterns, there can be only one name variable in a pattern, and we use the keyword # 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 in Section 3.2 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 [1] in the presence of mixins.)

5.2 Typing Judgments

The main typing rules of FMJ are presented in Figure 3, with auxiliary definitions presented in Figures 4, 5. *We recommend reading our text description and referring to the rules as needed in the flow of the text. Alternatively, the PDF version of this document contains hyperlinks in type rules, from operators to their text description.*

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 $\langle R_p, oR_f\rangle$, where R_p is the primary pattern, and oR_f the filter pattern. o can be + or -. We use $-o$ to reverse the sign: $-(+)$=-, $-(-)$=+.
  - R_p has the form $(T_1,\ <\overline{Y}\lhd\overline{P}>\overline{U}\rightarrow U_0)$. T_1 is the reflective type, over whose methods R_p iterates. $\overline{Y}$ are pattern type variables, bounded by $\overline{P}$, and $\overline{U}\rightarrow U_0$ the method pattern.
  - R_f has a similar form: $(T_2,\ \overline{V}\rightarrow V_0)$. However, note the lack of pattern type variables, due to the (formalism-only) simplification that the filter 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 $\langle R_p, oR_f\rangle$ tuple.

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

Expression typing:

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

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

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

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

Method typing:

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

$$\frac{\begin{array}{c}\Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}} \qquad \Gamma=\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}},\mathtt{this}\mapsto\mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \qquad \mathbb{M}_p=\mathtt{U}_0\ \mathtt{\#}\ (\overline{\mathtt{U}})\mathtt{:X}_i\mathtt{.methods} \qquad \mathbb{M}_f=\mathtt{V}_0\ \mathtt{\#}\ (\overline{\mathtt{V}})\mathtt{:X}_j\mathtt{.methods}\\ R_p=(\mathtt{X}_i,\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0) \qquad R_f=(\mathtt{X}_j,\ \overline{\mathtt{V}}\rightarrow\mathtt{V}_0) \qquad \Lambda=\langle R_p,R_f\rangle \qquad \Delta\vdash\overline{\mathtt{P}},\overline{\mathtt{U}},\mathtt{U}_0,\overline{\mathtt{V}},\mathtt{V}_0\ \mathit{ok}\\ \Delta;\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{S}_0 \qquad \Delta\vdash\mathtt{S}_0<:\mathtt{T}_0 \qquad CT(\mathtt{C})=\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{T}\ \{\ \ldots\ \} \qquad \Delta;\Lambda\vdash\mathit{override}(\mathtt{\#},\ \mathtt{T},\ \overline{\mathtt{T}}\rightarrow\mathtt{T}_0)\end{array}}{\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)}\ \mathtt{T}_0\ \mathtt{\#}\ (\overline{\mathtt{T}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\ \mathtt{OK\ IN}\ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}} \qquad \text{(T-METH-R)}$$

Class typing:

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

$$\frac{\begin{array}{ll} & \Delta=\overline{\mathtt{X}}<:\overline{\mathtt{N}} \qquad \Delta\vdash\overline{\mathtt{N}},\mathtt{T},\overline{\mathtt{T}}\ \mathit{ok} \qquad \overline{\mathfrak{M}}\ \mathtt{OK\ IN}\ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\\ \text{for all} & \mathfrak{M}_i,\mathfrak{M}_j\in\overline{\mathfrak{M}},\ \ \mathfrak{M}_i=\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>for(}\mathbb{M}_{p_i}\mathtt{;}o\mathbb{M}_{f_i}\mathtt{)}\ \mathtt{U}_0\ \mathtt{\#}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e}_i;\}\\ & \mathfrak{M}_j=\mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{Q}}\mathtt{>for(}\mathbb{M}_{p_j}\mathtt{;}o'\mathbb{M}_{f_j}\mathtt{)}\ \mathtt{V}_0\ \mathtt{\#}\ (\overline{\mathtt{V}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e}_j;\}\\ & \mathbb{M}_{p_i}=\mathtt{W}_i\ \mathtt{\#}\ (\overline{\mathtt{W}}_i)\mathtt{:X}_i\mathtt{.methods} \qquad \mathbb{M}_{f_i}=\mathtt{W}'_i\ \mathtt{\#}\ (\overline{\mathtt{W}}'_i)\mathtt{:X}'_i\mathtt{.methods}\\ & \mathbb{M}_{p_j}=\mathtt{W}_j\ \mathtt{\#}\ (\overline{\mathtt{W}}_j)\mathtt{:X}_j\mathtt{.methods} \qquad \mathbb{M}_{f_j}=\mathtt{W}'_j\ \mathtt{\#}\ (\overline{\mathtt{W}}'_j)\mathtt{:X}'_j\mathtt{.methods}\\ & R_{p_i}=(\mathtt{X}_i,\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{W}}_i\rightarrow\mathtt{W}_i) \qquad R_{f_i}=(\mathtt{X}'_i,\ \overline{\mathtt{W}}'_i\rightarrow\mathtt{W}'_i) \qquad \Lambda_i=\langle R_{p_i},oR_{f_i}\rangle\\ & R_{p_j}=(\mathtt{X}_j,\ \mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{W}}_j\rightarrow\mathtt{W}_j) \qquad R_{f_j}=(\mathtt{X}'_j,\ \overline{\mathtt{W}}'_j\rightarrow\mathtt{W}'_j) \qquad \Lambda_j=\langle R_{p_j},o'R_{f_j}\rangle\\ \text{implies} & \Delta\vdash\mathit{disjoint}(\Lambda_i,\ \Lambda_j)\end{array}}{\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{T}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathfrak{M}}\}\ \mathtt{OK}} \qquad \text{(T-CLASS-R)}$$

Well-formed types:

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

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

$$\frac{\begin{array}{c}CT(\mathtt{C})=\mathtt{class}\ \mathtt{C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\ \mathtt{T}\ \ \{\ \ldots\}\\ \Delta\vdash\overline{\mathtt{T}}\ \mathit{ok} \qquad \Delta\vdash\overline{\mathtt{T}}<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}}\end{array}}{\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \mathit{ok}} \qquad \text{(WF-CLASS)}$$

Figure 3. Typing Rules

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 subsumption and disjointness. Thus, we begin our discussion with a detailed explanation of the rules for subsumption and disjointness.

5.2.1 Subsumption and Disjointness

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

- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_{\Lambda}\Lambda'$. Range of Λ is subsumed by the range of Λ', under the assumptions of type environment Δ and the unifying type substitutions of $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.
- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_1\sqsubseteq_R R_2$. Single range R_1 is subsumed by single range R_2, under the assumptions of Δ and the unifying type substitutions of $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.
- $\Delta\vdash\mathit{disjoint}(\Lambda,\ \Lambda')$. The range of Λ and Λ' are disjoint under the assumptions of Δ.

Single range subsumption. In determining the subsumption between two reflective environments, we must first see how subsumption is determined between two single ranges. Rule SB-R (Figure 5) defines the two conditions for single range subsumption. 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 subsumption if the reflective base 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 reflective base set of R_1. We determine this relation using subtyping: a subtype's methods can be mapped onto its supertype's methods.(This is a source of incompleteness—see Section 6.) 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. Unification is defined by two relations:

- $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash$*unify*$(\mathtt{U}_0\mathtt{:}\overline{\mathtt{U}},\ \mathtt{V}_0\mathtt{:}\overline{\mathtt{V}})$. Rule UNI (Figure 5) 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`.

Method type lookup:

$$\frac{\Lambda=\langle R_p, oR_f\rangle \qquad R_p=(\texttt{X},\ \texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}_0)}{\Delta;\Lambda\vdash mtype(\texttt{\#},\ \texttt{X})=\overline{\texttt{U}}\rightarrow\texttt{U}_0} \qquad \text{(MT-VAR-R)}$$

$$\frac{\Lambda=\langle R_p, oR_f\rangle \qquad R_p=(\texttt{T},\ \texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{V}}\rightarrow\texttt{V}_0) \qquad \Delta;\Lambda\vdash mtype(\texttt{\#},\ bound_\Delta(\texttt{X}))=\overline{\texttt{U}}\rightarrow\texttt{U}_0}{\Delta;\Lambda\vdash mtype(\texttt{\#},\ \texttt{X})=\overline{\texttt{U}}\rightarrow\texttt{U}_0} \qquad \text{(MT-VAR-S)}$$

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

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

$$\frac{\begin{array}{c}CT(\texttt{C})=\texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{>}\lhd\texttt{T}\ \{\ldots\ \overline{\mathfrak{M}}\} \qquad \texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>for(}\mathbb{M}_p\texttt{;}o\mathbb{M}_f\texttt{)}\ \texttt{S}_0\ \texttt{\#}\ (\overline{\texttt{S}}\ \overline{\texttt{x}})\ \{\uparrow\texttt{e};\}\in\overline{\mathfrak{M}}\\ \mathbb{M}_p=\texttt{U}_0\ \texttt{\#}\ (\overline{\texttt{U}})\texttt{:X}_i\texttt{.methods} \quad \mathbb{M}_f=\texttt{V}_0\ \texttt{\#}\ (\overline{\texttt{V}})\texttt{:X}_j\texttt{.methods} \quad R_p=(\texttt{T}_i,\ [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}_0)) \quad R_f=(\texttt{T}_j,\ [\overline{\texttt{T}}/\overline{\texttt{X}}](\overline{\texttt{V}}\rightarrow\texttt{V}_0))\\ \Lambda_d=\langle R_p, oR_f\rangle \qquad \begin{cases}\Delta;\Lambda\vdash specialize(\texttt{m},\ \Lambda_d)=\Lambda_r & \text{if } \texttt{n}=\texttt{m}\\ \Lambda_r=\Lambda & \text{if } \texttt{n}=\#\end{cases} \qquad \Delta;[\overline{\texttt{W}}/\overline{\texttt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d\end{array}}{\Delta;\Lambda\vdash mtype(\texttt{n},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=[\overline{\texttt{T}}/\overline{\texttt{X}}][\overline{\texttt{W}}/\overline{\texttt{Y}}](\overline{\texttt{S}}\rightarrow\texttt{S}_0)} \qquad \text{(MT-CLASS-R)}$$

$$\frac{\begin{array}{ll} & CT(\texttt{C})=\texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{>}\lhd\texttt{T}\ \{\ldots\ \overline{\mathfrak{M}}\}\\ \text{for all} & \texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>for(}\mathbb{M}_p\texttt{;}o\mathbb{M}_f\texttt{)}\ \texttt{S}_0\ \texttt{\#}\ (\overline{\texttt{S}}\ \overline{\texttt{x}})\ \{\uparrow\texttt{e};\}\in\overline{\mathfrak{M}}\\ & \mathbb{M}_p=\texttt{U}_0\ \texttt{\#}\ (\overline{\texttt{U}})\texttt{:X}_i\texttt{.methods} \quad \mathbb{M}_f=\texttt{V}_0\ \texttt{\#}\ (\overline{\texttt{V}})\texttt{:X}_j\texttt{.methods}\\ & R_p=(\texttt{T}_i,\ [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}_0)) \quad R_f=(\texttt{T}_i,\ [\overline{\texttt{T}}/\overline{\texttt{X}}](\overline{\texttt{V}}\rightarrow\texttt{V}_0)) \quad \Lambda_d=\langle R_p, oR_f\rangle\\ & \begin{cases}\Delta;\Lambda\vdash specialize(\texttt{m},\ \Lambda_d)=\Lambda_r & \text{if } \texttt{n}=\texttt{m}\\ \Lambda_r=\Lambda & \text{if } \texttt{n}=\#\end{cases}\\ \text{implies} & \Delta\vdash disjoint(\Lambda_r,\Lambda_d)\end{array}}{\Delta;\Lambda\vdash mtype(\texttt{n},\ \texttt{C<}\overline{\texttt{T}}\texttt{>})=mtype(\texttt{m},\ [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T})} \qquad \text{(MT-SUPER-R)}$$

Specializing reflective environment:

$$\frac{\begin{array}{c}\Lambda_d=\langle R_p, oR_f\rangle \quad R_p=(\texttt{T}_i,\ \texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}) \quad R_f=(\texttt{T}_j,\ \overline{\texttt{V}}\rightarrow\texttt{V}) \quad \Delta;\Lambda\vdash mtype(\texttt{m},\ \texttt{T}_i)=\overline{\texttt{U}}'\rightarrow\texttt{U}' \quad R'_p=(\texttt{T}_i,\ \overline{\texttt{U}}'\rightarrow\texttt{U}')\\ \begin{cases}R'_f=(\texttt{T}_j,\ \overline{\texttt{V}}'\rightarrow\texttt{V}'_0) & o'=+ \quad \text{if} \quad \Delta;\Lambda\vdash mtype(\texttt{m},\texttt{T}_j)=\overline{\texttt{V}}'\rightarrow\texttt{V}'_0\\ \Delta;[\overline{\texttt{W}}/\overline{\texttt{Y}}]\vdash R'_p\sqsubseteq_R R_p \quad R'_f=[\overline{\texttt{W}}/\overline{\texttt{Y}}]R_f & o'=-o \quad \text{otherwise}\end{cases}\end{array}}{\Delta;\Lambda\vdash specialize(\texttt{m},\Lambda_d)=\langle R'_p, o'R'_f\rangle}$$

Valid method overriding:

$$\frac{\Delta\vdash validRange(\Lambda,\texttt{T}) \qquad \Delta;\Lambda\vdash mtype(\texttt{n},\ \texttt{T})=\overline{\texttt{V}}\rightarrow\texttt{V}_0 \ \text{ implies } \ \overline{\texttt{V}}=\overline{\texttt{U}} \quad \texttt{V}_0=\texttt{U}_0}{\Delta;\Lambda\vdash override(\texttt{n},\ \texttt{T},\ \overline{\texttt{U}}\rightarrow\texttt{U}_0)}$$

Field lookup:

$$\Delta\vdash \mathit{fields}(\texttt{Object})=\bullet \qquad\qquad \frac{CT(\texttt{C})=\texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{>}\lhd\texttt{T}\ \{\overline{\texttt{S}}\ \overline{\texttt{f}};\ \ldots\ \} \qquad \Delta\vdash \mathit{fields}(bound_\Delta([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{T}))=\overline{\texttt{D}}\ \overline{\texttt{g}}}{\Delta\vdash \mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>})=\overline{\texttt{D}}\ \overline{\texttt{g}},[\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}}\ \overline{\texttt{f}}}$$

Figure 4. Method type lookup, overriding and field lookup.

- $\Delta\vdash\texttt{T}\prec:_{\overline{\texttt{Z}}}\texttt{S}$ (Figure 5) indicates that type T is a valid substitution of S, i.e., it obeys the bound of S, using $\overline{\texttt{Z}}$ as pattern type variables.

Reflective (filtered) range subsumption. SB-Λ (Figure 5) defines the conditions for the range of reflective environment $\Lambda=\langle R_p, oR_f\rangle$ to be subsumed by the range of $\Lambda'=\langle R'_p, o'R'_f\rangle$. These conditions reflect precisely the informal rules of Section 3.2. First, regardless of filter patterns, the *primary* range of Λ should at least be subsumed by the primary range of Λ'. Secondly, for every method in R_p that satisfies the filter pattern oR_f, the corresponding method in R'_p should satisfy the filter pattern $o'R'_f$. There are a couple of ways to guarantee oR_f *implies* $o'R'_f$. If +R_f is true, and R_f is subsumed by R'_f, then +R'_f is also true (i.e., if there is at least one method in R_f, then there is at least one method in a larger range, R'_f). This condition is expressed by $\Delta;\bullet\vdash R_f\sqsubseteq_R[\overline{\texttt{W}}/\overline{\texttt{Y}}]R'_f$, if $o=o'=+$. We apply the unifying type substitutions for the primary ranges to the filter range R'_f: in order to properly compare the ranges of R_f and R'_f, we need to restrict R'_f 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_f is subsumed by $[\overline{\texttt{W}}/\overline{\texttt{Y}}]R'_f$. This is because filter patterns do not have pattern type variables of their own, and pattern type variables from the primary pattern are treated as constants in the filter patterns. Similarly, if -R_f is true, and R_f subsumes R'_f, -R'_f is also true.

Reflective range disjointness. Disjointness of reflective ranges is defined by rules DS-Λ1 and DS-Λ2. DS-Λ1 specifies the conditions for disjointness when Λ and Λ' reflect over types from the same subtyping hierarchy ($\Delta\vdash\texttt{T}_p<:\texttt{T}'_p$ or $\Delta\vdash\texttt{T}'_p<:\texttt{T}_p$). In this case, Λ and Λ' are disjoint if their primary ranges are disjoint. However, if the two primary ranges *do* have overlap, (i.e., $\Delta;[\overline{\texttt{W}}/\overline{\texttt{Z}}]\vdash unify(\texttt{U}_0\texttt{:}\overline{\texttt{U}},\ \texttt{U}'_0\texttt{:}\overline{\texttt{U}}')$: a *two-way* unification exists between the primary ranges) it is still possible for Λ and Λ' to be disjoint if we can establish that for the methods that fall into the overlap, the filter patterns cannot be satisfied simultaneously. There are two ways to establish the exclusivity of two filter patterns. First, if +R_f is true, and R_f is subsumed by R'_f, then -R'_f cannot possibly by true. Similarly, if +R'_f is true, and R'_f is subsumed by R_f, then -R_f cannot be true.

DS-Λ2 specifies a different condition for disjointness: if the primary range of Λ, R_p, can be subsumed by the filter range of Λ', R'_f, and the filtering condition is negative (i.e., -R'_f), then it is guaranteed that Λ and Λ' have disjoint names. The reasoning is that any method matched by the primary range R'_p is guaranteed to *not* satisfy the filter pattern -R_f, thus the two filtered ranges are disjoint.

Reflective range subsumption:

$$\frac{\begin{array}{c}\Lambda{=}\langle R_p,oR_f\rangle \quad \Lambda'{=}\langle R'_p,o'R'_f\rangle \quad R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\rightarrow\mathtt{V}_0) \quad R_f{=}(\mathtt{T}_f,\ \overline{\mathtt{U}}'\rightarrow\mathtt{U}'_0) \quad R'_f{=}(\mathtt{T}'_f,\ \overline{\mathtt{V}}'\rightarrow\mathtt{V}'_0) \\ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_p\sqsubseteq_R R'_p \qquad \begin{cases}\Delta;\bullet\vdash R_f\sqsubseteq_R[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_f & \text{if } o=o'=+\\ \Delta;\bullet\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]R'_f\sqsubseteq_R R_f & \text{if } o=o'=-\end{cases}\end{array}}{\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'} \qquad \text{(SB-}\Lambda\text{)}$$

Single range subsumption:

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

Reflective range disjointness:

$$\frac{\begin{array}{c}\Lambda{=}\langle R_p,oR_f\rangle \quad \Lambda'{=}\langle R'_p,o'R'_f\rangle \quad R_p{=}(\mathtt{T}_p,\ \mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0) \quad R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}'\rightarrow\mathtt{U}'_0) \\ R_f{=}(\mathtt{T}_f,\ \overline{\mathtt{V}}\rightarrow\mathtt{V}_0) \quad R'_f{=}(\mathtt{T}'_f,\ \overline{\mathtt{V}}'\rightarrow\mathtt{V}'_0) \quad \Delta\vdash\mathtt{T}_p{<:}\mathtt{T}'_p \text{ or } \Delta\vdash\mathtt{T}'_p{<:}\mathtt{T}_p \quad \Delta'{=}\Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{Q}},\overline{\mathtt{Y}}{<:}\overline{\mathtt{P}} \quad \overline{\mathtt{Z}}{=}\overline{\mathtt{X}},\overline{\mathtt{Y}} \\ \text{for all } \overline{\mathtt{W}},\ \Delta';[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]\vdash\mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{U}'_0{:}\overline{\mathtt{U}}') \text{ implies } \begin{cases}\Delta';\bullet\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R_f\sqsubseteq_R[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R'_f & \text{if } o=+,o'=-\\ \Delta';\bullet\vdash[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R'_f\sqsubseteq_R[\overline{\mathtt{W}}/\overline{\mathtt{Z}}]R_f & \text{if } o=-,o'=+\end{cases}\end{array}}{\Delta\vdash\mathit{disjoint}(\Lambda,\ \Lambda')} \qquad \text{(DS-}\Lambda 1\text{)}$$

$$\frac{\begin{array}{ll}\Lambda{=}\langle R_p,oR_f\rangle & R_p{=}(\mathtt{T}_p,\ \mathtt{<}\overline{\mathtt{X}}\lhd\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)\\ \Lambda'{=}\langle R'_p,o'R'_f\rangle & R'_p{=}(\mathtt{T}'_p,\ \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}\rightarrow\mathtt{V}_0)\end{array} \qquad \begin{cases}\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R'_p\sqsubseteq_R R_f & o=- \qquad \text{or}\\ \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash R_p\sqsubseteq_R R'_f & o'=-\end{cases}}{\Delta\vdash\mathit{disjoint}(\Lambda,\ \Lambda')} \qquad \text{(DS-}\Lambda 2\text{)}$$

Subtype range validity:

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

$$\frac{\begin{array}{ll} & CT(\mathtt{C}){=}\mathtt{class\ C<}\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}\mathtt{>}\lhd\mathtt{S}\ \{\ \ldots\ \overline{\mathfrak{M}}\} \qquad \Delta\vdash\mathit{validRange}(\Lambda,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S})\\ \text{for all} & \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>for(}\mathbb{M}_p\mathtt{;}\,o\mathbb{M}_f\mathtt{)\ldots}\in\overline{\mathfrak{M}} \quad \mathbb{M}_p{=}\mathtt{V}_0\ \mathtt{\#\ (}\overline{\mathtt{U}}\mathtt{):X}_i\mathtt{.methods} \quad \mathbb{M}_f{=}\mathtt{V}_0\ \mathtt{\#\ (}\overline{\mathtt{V}}\mathtt{):X}_j\mathtt{.methods}\\ & R_p{=}(\mathtt{X}_i,\ \mathtt{<}\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0) \quad R_f{=}(\mathtt{X}_j,\ \overline{\mathtt{V}}\rightarrow\mathtt{V}_0) \quad \Lambda'{=}[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\langle R_p,oR_f\rangle)\\ \text{implies} & \Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda' \text{ or } \Delta\vdash\mathit{disjoint}(\Lambda',\ \Lambda)\end{array}}{\Delta\vdash\mathit{validRange}(\Lambda,\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})} \qquad \text{(VR-CLASS)}$$

Type Unification:

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

Subtyping rules:

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

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

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

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

Pattern matching rules:

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

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

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

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

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

Figure 5. Reflection related auxiliary functions.

Similarly, if R_p is subsumed by R_f, and -R'_f is the filter condition, disjointness is also established.

These rules reflect very closely the informal rules of Section 3.2 modulo the small differences in the formalism mentioned in Section 5.1: 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.

5.2.2 Valid Method Invocation

The rest of the typing rules add the machinery to standard FGJ type checking to express checks using range subsumption and disjointness. For instance, to determine valid method invocation is to determine that the reflective environment of the invocation is subsumed by the reflective environment of the declaration. T-INVK (Figure 3) specifies conditions for a well-typed method invocation. It relies on $\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{n},\ \mathtt{T})$ (Figure 4) 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-R says that the type of method with a variable name `#` in a type `T`, where `T` is a subtype of the reflective type for the current reflective environment, Λ, is exactly the type specified by the primary pattern of Λ. Otherwise, if `T` is a type variable, then we must look for the method type in its bound (MT-VAR-S).

MT-CLASS-R lists conditions for retrieving the type of `n` in $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$, where $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$ has reflectively declared methods. If `n` is a name variable, this is simply checking that the range of reference, which is the current reflective environment, is subsumed by the declaration reflective environment. When `n` is a constant name `m`, however, we need to check whether `m` is within the range of method names in the declaration reflective range. We do so by *specializing* the declaration reflective environment using `m`. Specializing a range based on a constant name is just a way to package the information for uniform use by our subsumption and disjointness checks (which apply to entire ranges with patterns and not single methods). The main property of $\mathit{specialize}(\mathtt{m},\ \Lambda_d){=}\Lambda_r$ is that `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 subsumed by Λ_d. Similarly, `m` does not belong in Λ_d, if and only if Λ_r is disjoint from Λ_d.

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, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]$.

5.2.3 Uniqueness of Definitions

T-METH-R (Figure 3) 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 4). $override(\mathtt{n},\ \mathtt{T},\ \overline{\mathtt{U}}{\to}\mathtt{U}_0)$ determines whether method `n`, defined in some *subclass* of `T` with type signature $\overline{\mathtt{U}}{\to}\mathtt{U}_0$, properly overrides method `n` in `T`. If method `n` exists in `T`, it must have the exact same argument and return types as `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 5.) Additionally, the reflective range of `n` in the subclass must be either completely subsumed by one of `T`'s reflective ranges, or disjoint from all the reflective ranges of `T` (and, transitively, `T`'s superclasses). This condition is enforced using $\Delta{\vdash}validRange(\Lambda,\ \mathtt{T})$ (Figure 5).

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.

5.3 Soundness:

We prove the soundness of FMJ by proving Subject Reduction and Progress. (Our semantics and proofs are in the appendix.)

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

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

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`.

6. Completeness and (Limits of) Decidability

Since morphing adds explicit iteration at compile-time, it naturally flirts with undecidability. Indeed, the unrestricted question of range subsumption for static iterators is undecidable. Of course, the undecidability concerns the *complete* version of the decision problem, i.e., telling exactly when (without underapproximating) a range is subsumed by another. We can encode undecidable decision problems (e.g., Datalog query containment, or equivalence of arbitrary recursive functions) in extended versions of the MJ type system. We do not yet have a precise understanding of which language features allow reducing undecidable problems to type-checking MJ programs. We have, however, found that any of three simple extensions result in this kind of expressiveness: We can allow unrestricted nesting of static-`for` statements; or allow type variables bound only in a filter pattern to be used in the declared code; or allow free type variables in the type we iterate over (e.g., "`<T,Y> T f : C<Y>.fields`" where `Y` is not bound). We do not present these undecidability results due to space limitations, but their main structure is not surprising. For instance, if we allow type variables bound only in the filter pattern to appear in the declared code, then the filter pattern becomes a full-fledged iteration. This lets us encode (for instance) arithmetic with addition and multiplication, or relational algebra (with a static-`for` construct to implement the join operator).

Nevertheless, MJ admits a decidable type-checking process without sacrificing the expressiveness we need in practice. Note that the MJ language, as defined earlier, already places structural limits on what is expressible (e.g., static-`for`s cannot be nested). To establish the decidability of our type system, we also need some type-level restrictions, which can be viewed as sources for incompleteness: First, we use subtyping to compare types that we iterate over, for the purposes of checking range subsumption. This is conservative, since two classes unrelated by subtyping could have the same range of methods, and this could have been established through other reasoning in our type system. (For disjointness we do not make the same approximation, instead assuming that ranges *may* conflict, unless we can prove otherwise.) Second, we enforce limitations on possible circularities in either subtyping or static iteration cross-type references.

To avoid subtyping circularities we inherit a technique from standard mixin-based type systems. Applying the same restrictions as Allen et al. [1] (i.e., a declared supertype cannot be a type with a mixin superclass itself), we can guarantee that there is no cyclic inheritance in FMJ, and thus, subtyping is decidable.

Another source for non-termination in FMJ is in circularly dependent method definitions. For example,

```
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>>`! This type of definition would cause infinite recursion in the derivation of *mtype*:

$$\begin{array}{cl}
\ldots & \\
\Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{D<T>}) & \text{MT-CLASS-R and } \textit{specialize} \\ \hline
\Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{C<D<T>>}) & \text{MT-SUPER-S} \\ \hline
\Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{D<T>}) & \text{MT-SUPER-S} \\ \hline
\Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{T}) & \text{MT-CLASS-R and } \textit{specialize} \\ \hline
\Delta;\Lambda{\vdash}mtype(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) &
\end{array}$$

We detect such circularity by constructing a chain of reflective *reachability*. The chain of reachability for a type `T` is essentially all the types $mtype(\mathtt{n},\mathtt{T})$ could recursively call upon. For example, the chain of reachability for the above `C<X>` is `C<X>`, `X`, `D<X>`, `C<D<X>>`, ... We stop the chain construction as soon as we see a re-occurrence of any type already in the chain, in *any* form of instantiation. We reject classes with such circular dependency. Since there is a finite number of classes, the chain must either see a reoccurrence of some class, or be finitely sized. The length of the chain serves as a measure function for each call of *mtype*. The finite size of the chain means the measure function cannot decrease infinitely, thus proving termination. A more sophisticated protocol would be possible, to make the check less conservative, but we have yet to encounter a plausible use that needs it.

One more interesting question concerns the completeness of our range subsumption and disjointness logic. For instance, do the rules in Section 3.2 miss legal instances? We have shown some results in this direction, which we only summarize here due to space limitations. Our rules are complete for programs without filter patterns. For programs with filter patterns, our rules are complete under some extra assumptions of independence of the types that the primary and filter pattern iterate over.

7. Related Work and Conclusions

The main capabilities of morphing can typically be emulated only with much lower-level mechanisms, such as reflection and meta-object protocols [14], aspect-oriented programming [15], or pattern-based program generation and transformation [2, 3, 20]. The main goal of our work is to promote these abilities to high-level language features, with full modular type-safety. None of the above mechanisms offer such safety guarantees: a transform, aspect, or meta-class cannot be type-checked independently from the rest of the program, in a way that guarantees it is type-correct for all possible uses it may be employed in.

An interesting special case of program generation is *staging languages* such as MetaML [19] and MetaOCaml [5]. 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. Generally, staging languages target program specialization rather than full program generation: the program must remain valid even when staging annotations are removed. It is interesting that even recent metaprogramming tools, such as Template Haskell [18] are explicitly not modularly type safe—its authors acknowledge that they sacrifice the MetaML guarantees for expressiveness.

The closest related work to MJ is represented by mechanisms such as type-safe reflection (Genoupe) [6], compile-time reflection (CTR) [8], and safe program generation (SafeGen) [10]. Yet none of these mechanisms offer MJ's expressiveness and full modular type checking guarantees. For instance, the Genoupe [6] approach has been shown unsafe, as the reasoning depends on properties that can change at runtime; CTR [8] only captures undefined variable and type incompatibility errors (not duplicate definitions), does not offer a formal system or proof of soundness, and has less expressiveness compared to MJ (especially with respect to matching method arguments); SafeGen [10] has no soundness proof and relies on the capabilities of an automatic theorem prover—an unpredictable and unfriendly process from the programmer's perspective. Additionally, MJ integrates smoothly with the concept of generic classes in a programming language, as opposed to defining a "transformation" as a top-level concept.

The body of work in static type-directed programming [13, 22] is similar to MJ in that such mechanisms allow fields and methods of a data type to be traversed without being explicitly named. However, these techniques fall short of MJ (and static reflection work in general [6, 8, 10]) in that they do not allow *declarations* to be made using names or types retrieved by inspecting data types.

MJ can be viewed as a tool for class refinement or extension. In this sense, MJ is related to work on virtual classes [7] and other modular class expansion and refinement techniques [17, 21]. However, MJ's goal is orthogonal to such work. MJ provides a way to write *generic* refinements and extensions—refinements that can be applied to different types through type-instantiation. MJ's main concern is not the technique used to integrate these refinements into existing classes. Indeed the main MJ insights can be combined with approaches such as *expanders* [21], to integrate generic (but modularly type-safe) components into existing class hierarchies.

Overall, we view morphing as *the disciplined kernel of metaprogramming*. We believe that its inclusion in mainstream languages will be a topic of major importance for decades to come and filter patterns represent a big step forward in this direction.

References

[1] E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2003.

[2] J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2001.

[3] J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. of Programming Language Design and Implementation (PLDI)*, 2002.

[4] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP'90: Proc. of the European Conference on Object-Oriented programming and Object Oriented Programming, Systems, Languages, and Applications*, 1990.

[5] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2003.

[6] D. Draheim, C. Lutteroth, and G. Weber. A type system for reflective program generators. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2005.

[7] E. Ernst, K. Ostermann, and W. R. Cook. A virtual class calculus. In *Proc. of Principles of Programming Languages (POPL)*, 2006.

[8] M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2006.

[9] S. S. Huang and Y. Smaragdakis. Easy language extension with Meta-AspectJ. In *Proc. of International Conference on Software Engineering (ICSE)*, 2006.

[10] S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Proc. of Generative Programming and Component Engineering (GPCE)*, 2005.

[11] S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, 2007.

[12] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1999.

[13] P. Jansson and J. Jeuring. PolyP - a polytypic programming language extension. In *Proc. of Principles of Programming Languages (POPL)*, 1997.

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

[15] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of European Conf. on Object-Oriented Programming (ECOOP)*, 1997.

[16] M. Mohnen. Interfaces with default implementations in Java. In *Proc. of Principles and Practice of Programming*, 2002.

[17] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[18] T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proc. of the ACM SIGPLAN workshop on Haskell*, 2002.

[19] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. of Partial Evaluation and semantics-based Program Manipulation (PEPM)*, 1997.

[20] E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In *Domain-Specific Program Generation*. Springer-Verlag, 2004. LNCS 3016.

[21] A. Warth, M. Stanojevic, and T. Millstein. Statically scoped object adaptation with expanders. In *Proc. of Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2006.

[22] S. Weirich and L. Huang. A design for type-directed java. In *Workshop on Object-Oriented Developments (WOOD)*, 2004.

Appendix

Reduction Rules:

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

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

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

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

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

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

Figure 7. Reduction Rules

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

Proof. Prove by structural induction on the reduction rules.

Case R-FIELD:
By R-FIELD, T-NEW, T-FIELD,
$\mathtt{e}=\mathtt{new\ C<\overline{T}>(\overline{e}).f}_i$ $\quad$ $\mathtt{e}'=\mathtt{e}_i$
$\Delta;\Lambda;\Gamma\vdash\mathtt{new\ C<\overline{T}>(\overline{e})}\in\mathtt{C<\overline{T}>}$ $\quad$ $\mathit{bound}_\Delta(\mathtt{C<\overline{T}>})=\mathtt{C<\overline{T}>}$
$\Delta\vdash\mathit{fields}(\mathtt{C<\overline{T}>})=\overline{\mathtt{U}}\,\overline{\mathtt{f}}$ $\quad$ $\Delta;\Lambda;\Gamma\vdash\mathtt{new\ C<\overline{T}>(\overline{e}).f}_i\in\mathtt{U}_i$
$\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$ $\quad$ $\Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{U}}$
Thus, $\Delta\vdash\mathtt{e}_i\in\mathtt{S}_i$, and $\Delta\vdash\mathtt{S}_i<:\mathtt{U}_i$.
Case R-INVK:
By R-INVK, T-NEW, T-INVK
$\mathtt{e}=\mathtt{new\ C<\overline{T}>(\overline{e}).n(\overline{d})}$ $\quad$ $\mathtt{e}'=[\overline{\mathtt{d}}/\overline{\mathtt{x}},\mathtt{new\ C<\overline{T}>/this}]\mathtt{e}_0$
$\mathit{mbody}(\mathtt{n},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\mathtt{e}_0)$ $\quad$ $\Delta\vdash\mathtt{new\ C<\overline{T}>(\overline{e})}\in\mathtt{C<\overline{T}>}$
$\Delta\vdash\mathtt{C<\overline{T}>}\ \mathit{ok}$ $\quad$ $\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{n},\ \mathtt{C<\overline{T}>})=\overline{\mathtt{T}}'\rightarrow\mathtt{T}$
$\Delta;\Gamma;\Lambda\vdash\overline{\mathtt{d}}\in\overline{\mathtt{S}}$ $\quad$ $\Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}'$
By Lemma 1, for some $\mathtt{S}$, $\Delta\vdash\mathtt{S}<:\mathtt{T}$, $\Delta\vdash\mathtt{S}\ \mathit{ok}$, $\Delta;\Lambda;\overline{\mathtt{x}}:\overline{\mathtt{T}}',\mathtt{this}:\mathtt{C<\overline{T}>}\vdash\mathtt{e}_0\in\mathtt{S}$.
By Lemma 3 and Lemma 4, we have $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}},\mathtt{new\ C<\overline{T}>/this}]\mathtt{e}_0\in\mathtt{S}$.
Case RC-FIELD: $\mathtt{e}=\mathtt{e}_0\mathtt{.f}$, $\mathtt{e}'=\mathtt{e}_0'\mathtt{.f}$
By T-FIELD, and induction hypothesis,
$\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\mathtt{.f}\in\mathtt{T}_i$ $\quad$ $\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0$
$\Delta\vdash\mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0))=\overline{\mathtt{T}}\,\overline{\mathtt{f}}$
$\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0'\in\mathtt{S}_0$ $\quad$ $\Delta\vdash\mathtt{S}_0<:\mathtt{T}_0$
By Lemma 10, $\Delta\vdash\mathit{fields}(\mathit{bound}_\Delta(\mathtt{S}_0))=\overline{\mathtt{S}}\,\overline{\mathtt{g}}$, $\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0'\mathtt{.f}\in\mathtt{S}_i$, $\mathtt{S}_i=\mathtt{T}_i$. By S-REFL, $\Delta\vdash\mathtt{S}_i<:\mathtt{T}_i$.
Case RC-INV-RECV: $\mathtt{e}=\mathtt{e}_0\mathtt{.n(\overline{e})}$, $\mathtt{e}'=\mathtt{e}_0'\mathtt{.n(\overline{e})}$
By T-INVK,
$\Delta;\Lambda;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0$ $\quad$ $\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{T}}'$
$\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{n},\,\mathtt{T}_0)=\overline{\mathtt{T}}\rightarrow\mathtt{T}$ $\quad$ $\Delta\vdash\overline{\mathtt{T}}'<:\overline{\mathtt{T}}$
By induction hypothesis, Lemma 14, and T-INVK,
$\Delta;\Lambda;\Gamma\vdash\mathtt{e}'\in\mathtt{S}_0$ $\quad$ $\Delta\vdash\mathtt{S}_0\in\mathtt{T}_0$
$\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{n},\,\mathtt{S}_0)=\overline{\mathtt{T}}\rightarrow\mathtt{T}$ $\quad$ $\Delta;\Lambda;\Gamma\vdash\mathtt{e}'\mathtt{.n(\overline{e})}\in\mathtt{T}$

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

Proof. 1. Easy using the well-typedness of subexpression and T-FIELD.

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

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* $\mathtt{v}$ *such that* $\emptyset;\emptyset;\emptyset\vdash\mathtt{v}\in\mathtt{S}$ *and* $\emptyset\vdash\mathtt{S}<:\mathtt{T}$ *for some type* $\mathtt{S}$.

Proof. Conclusion is obvious from Theorem 1 and Theorem 2 □

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

Proof. By induction on the derivation of $\mathit{mbody}(\mathtt{n},\ \mathtt{C<\overline{T}>})=(\overline{\mathtt{x}},\mathtt{e}_0)$:

Case MB-CLASS-S:
$\mathit{CT}(\mathtt{C})=\mathtt{class\ C<\overline{X}\triangleleft\overline{N}>\triangleleft N}\ \{\ldots\overline{\mathtt{M}}\}$
$\mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\in\overline{\mathtt{M}}$ $\quad$ $\mathtt{e}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}$
By MT-CLASS-S,
$\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{m},\,\mathtt{C<\overline{T}>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)$
$\overline{\mathtt{S}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{U}}$ $\quad$ $\mathtt{S}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0$
By T-METH-S,
$\overline{\mathtt{X}}<:\overline{\mathtt{N}};\emptyset;\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C<\overline{X}>}\vdash\mathtt{e}\in\mathtt{U}_0'$ $\quad$ $\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{U}_0'<:\mathtt{U}_0$
By Lemma 7 and 4,
$\Delta;\Lambda;[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}},\mathtt{this}\mapsto\mathtt{C<\overline{X}>})\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'$
By Lemma 5 and 4, $\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0$.
Let $\mathtt{S}'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{U}_0'$.
Case MB-CLASS-R:
$\mathit{CT}(\mathtt{C})=\mathtt{class\ C<\overline{X}\triangleleft\overline{N}>\triangleleft T}\ \{\ \ldots\ \overline{\mathfrak{M}}\}$
$\mathtt{<\overline{Y}\triangleleft\overline{P}>for}(\mathbb{M}_p\,;o\mathbb{M}_f)\ \mathtt{S}''\ \mathtt{*}\ (\overline{\mathtt{S}}''\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e}_0';\}\in\overline{\mathfrak{M}}$
$\mathbb{M}_p=\mathtt{U\ *\ (\overline{U}):X}_i\mathtt{.methods}$ $\quad$ $\mathbb{M}_f=\mathtt{V\ *\ (\overline{V}):X}_j\mathtt{.methods}$
$R_p=(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<\overline{Y}\triangleleft\overline{P}>}\overline{\mathtt{U}}\rightarrow\mathtt{U}))$ $\quad$ $R_f=(\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\rightarrow\mathtt{V}))$
$\Lambda_d=\langle R_p,R_f\rangle$ $\quad$ $\Lambda_r=\overline{\mathtt{Y}}<:\overline{\mathtt{P}};\emptyset\vdash\mathit{specialize}(\mathtt{m},\ \Lambda_d)$
$\overline{\mathtt{Y}}<:\overline{\mathtt{P}};[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$ $\quad$ $\mathtt{e}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\mathtt{*}]\mathtt{e}_0'$
By MT-CLASS-R, and Lemma 2,
$\Delta;\Lambda\vdash\mathit{mtype}(\mathtt{n},\ \mathtt{C<\overline{T}>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{S}}''\rightarrow\mathtt{S}'')$
$\overline{\mathtt{S}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{S}}''$ $\quad$ $\mathtt{S}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{S}''$
By T-METH-R and T-CLASS-R,
$\Delta''=\overline{\mathtt{X}}<:\overline{\mathtt{N}};\overline{\mathtt{Y}}<:\overline{\mathtt{P}}$ $\quad$ $R_p''=(\mathtt{X}_i,\ \overline{\mathtt{U}}\rightarrow\mathtt{U})$ $\quad$ $R_f''=(\mathtt{X}_j,\ \overline{\mathtt{V}}\rightarrow\mathtt{V})$
$\Lambda''=\langle R_p'',R_f''\rangle$ $\quad$ $\Delta'';\Lambda'';\Gamma''\vdash\mathtt{e}_0'\in\mathtt{T}_0$ $\quad$ $\Delta\vdash\mathtt{T}_0<:\mathtt{S}''$
By WF-CLASS, $\Delta\vdash\overline{\mathtt{T}}<:\overline{\mathtt{N}}$. And by Lemma 5, 7, and 4,
$\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_0<:[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}''$
$\Delta;[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda'';\overline{\mathtt{x}}\mapsto[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}'',\mathtt{this}\mapsto\mathtt{C<\overline{T}>}\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{e}_0'\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}_0$
By construction, $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\Lambda=\Lambda_d$.
By Lemma 6, 11, 15,
$\Delta;\Lambda_r;\overline{\mathtt{x}}\mapsto\overline{\mathtt{S}}\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\mathtt{m}/\mathtt{*}]\mathtt{e}_0'\in[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0$
By Lemma 6, $\Delta\vdash[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}_0<:\mathtt{S}$
Case MB-SUPER-S, MB-SUPER-R:
Easy by induction hypothesis.

Method body lookup:

$$\frac{CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N}\ \{\ldots\ \overline{\mathtt{M}}\}\qquad \mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e;}\}\in\overline{\mathtt{M}}}{mbody(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{x}},\mathtt{e})}\quad\text{(MB-CLASS-S)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{T}\ \{\ldots\ \overline{\mathfrak{M}}\}\qquad \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)}\ \mathtt{S}_0\ \mathtt{*}\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e;}\}\in\overline{\mathfrak{M}}\\ \mathbb{M}_p=\mathtt{U}_0\ \mathtt{*}\ (\overline{\mathtt{U}})\mathtt{:X}_i\mathtt{.methods}\quad \mathbb{M}_f=\mathtt{V}_0\ \mathtt{*}\ (\overline{\mathtt{V}})\mathtt{:X}_j\mathtt{.methods}\quad R_p=(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0))\quad R_f=(\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\rightarrow\mathtt{V}_0))\\ \Lambda_d=\langle R_p,oR_f\rangle\qquad \overline{\mathtt{Y}}<:\overline{\mathtt{P}};\emptyset\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r\qquad \overline{\mathtt{Y}}<:\overline{\mathtt{P}};[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d\end{array}}{mbody(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{W}}/\overline{\mathtt{Y}}](\overline{\mathtt{x}},[\mathtt{m}/\mathtt{*}]\mathtt{e})}\quad\text{(MB-CLASS-R)}$$

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

$$\frac{\begin{array}{ll} & CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{T}\ \{\ldots\ \overline{\mathfrak{M}}\}\\ \text{for all} & \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>for(}\mathbb{M}_p\mathtt{;}o\mathbb{M}_f\mathtt{)}\ \mathtt{S}_0\ \mathtt{*}\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e;}\}\in\overline{\mathfrak{M}}\\ & \mathbb{M}_p=\mathtt{U}_0\ \mathtt{*}\ (\overline{\mathtt{U}})\mathtt{:X}_i\mathtt{.methods}\quad \mathbb{M}_f=\mathtt{V}_0\ \mathtt{*}\ (\overline{\mathtt{V}})\mathtt{:X}_j\mathtt{.methods}\\ & R_p=(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0))\quad R_f=(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\rightarrow\mathtt{V}_0))\quad \Lambda_d=\langle R_p,oR_f\rangle\\ & \overline{\mathtt{Y}}<:\overline{\mathtt{P}};\emptyset\vdash specialize(\mathtt{m},\Lambda_d)=\Lambda_r\\ \text{implies} & \overline{\mathtt{Y}}<:\overline{\mathtt{P}}\vdash disjoint(\Lambda_r,\ \Lambda_d)\end{array}}{mbody(\mathtt{m},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=mbody(\mathtt{m},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})}\quad\text{(MB-SUPER-R)}$$

Figure 6. Method body lookup rules.

□

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

$\Lambda_1=\langle R_{p_1},oR_{f_1}\rangle\qquad R_{p_1}=(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)$

$\Lambda_2=\langle R_{p_2},o'R_{f_2}\rangle\qquad R_{p_2}=(\mathtt{T}_2,\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}'\rightarrow\mathtt{U}_0')$

For any Λ_3 *where*

$\Lambda_3=\langle R_{p_3},o''R_{f_3}\rangle\qquad R_{p_3}=(\mathtt{T}_3,\ \mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\overline{\mathtt{V}}\rightarrow\mathtt{V}_0)$

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

Proof. We prove by inspecting the two rules for disjointness.

Case DS-Λ1:
We prove by contradiction. Let there be some $\overline{\mathtt{W}}'$ such that $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$
By SB-Λ and SB-R, we have:
$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_{p_3}\sqsubseteq_R R_{p_1}\qquad \Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}}\vdash unify(\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}},\ \mathtt{U}_0\mathtt{:}\overline{\mathtt{U}})$
$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{p_3}\sqsubseteq_R R_{p_2}\qquad \Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\vdash unify(\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}},\ \mathtt{U}_0'\mathtt{:}\overline{\mathtt{U}}')$
By UNI,
$[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}=[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{U}_0\mathtt{:}\overline{\mathtt{U}}\qquad$ for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}}\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{X}}}\mathtt{X}_i$
$[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{U}_0'\mathtt{:}\overline{\mathtt{U}}'\qquad$ for all $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\vdash\mathtt{W}_i'\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$
Since neither $\overline{\mathtt{X}}$ or $\overline{\mathtt{Y}}$ appear in $\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}$,
$[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}=\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}\qquad [\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{U}_0\mathtt{:}\overline{\mathtt{U}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{U}_0'\mathtt{:}\overline{\mathtt{U}}'$
By Lemma 4 and PM-* rules,
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 $\mathtt{Y}_i\in\overline{\mathtt{Y}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}}\vdash\mathtt{W}_i'\prec:_{\overline{\mathtt{X}}:\overline{\mathtt{Y}}}\mathtt{Y}_i$,
Then clearly,
$\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{P}},\overline{\mathtt{Y}}<:\overline{\mathtt{Q}};[(\overline{\mathtt{W}}\mathtt{:}\overline{\mathtt{W}}')/(\overline{\mathtt{X}}\mathtt{:}\overline{\mathtt{Y}})]\vdash unify(\mathtt{U}_0\mathtt{:}\overline{\mathtt{U}},\ \mathtt{U}_0'\mathtt{:}\overline{\mathtt{U}}')$
By DS-Λ1, for $\Delta\vdash disjoint(\Lambda_1,\ \Lambda_2)$, o and o' must have opposite signs. But by SB-Λ, for Λ to be subsumed by Λ_1, o'' must be the same as o. Similarly, for Λ to be subsumed by Λ_2, o'' must be the same as o'. This means o and o' must have the same signs. Thus we arrive at the contradiction.

Case DS-Λ2:
For some $\overline{\mathtt{W}}''$, $\Delta;[\overline{\mathtt{W}}''/\overline{\mathtt{X}}]\vdash R_{p_2}\sqsubseteq_R R_{f_1}$, $o{=}-$
Let $R_{f_1}=(\mathtt{T}_1',\ \overline{\mathtt{V}}'\rightarrow\mathtt{V}_0')$, then by SB-$R$ and UNI,
$\mathtt{U}_0'\mathtt{:}\overline{\mathtt{U}}'=[\overline{\mathtt{W}}''/\overline{\mathtt{X}}]\mathtt{V}_0'\mathtt{:}\overline{\mathtt{V}}'\qquad$ for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta;\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\vdash\mathtt{W}_i'\prec:_{\overline{\mathtt{X}}}\mathtt{X}_i$
Since $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_1$, by SB-Λ,
$\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_{p_3}\sqsubseteq_R R_{p_1}\qquad \Delta;\bullet\vdash[\overline{\mathtt{W}}/\overline{\mathtt{X}}]R_{f_1}\sqsubseteq_R R_{f_3}\qquad o''{=}-$
Assume that $\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash\Lambda_3\sqsubseteq_\Lambda\Lambda_2$.
By SB-Λ,
$\Delta;[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\vdash R_{p_3}\sqsubseteq_R R_{p_2}\qquad \Delta;\bullet\vdash[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]R_{f_2}\sqsubseteq_R R_{f_3}\qquad o'{=}-$
Let $R_{p_3}=(\mathtt{T}_3,\ \overline{\mathtt{S}}\rightarrow\mathtt{S}_0)$. By SB-$R$ and UNI,
$\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}]\mathtt{U}_0'\mathtt{:}\overline{\mathtt{U}}'\qquad$ for all $\mathtt{X}_i\in\overline{\mathtt{X}}$, $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{Q}},\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\vdash\mathtt{W}_i'\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$
Then, $\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}=[\overline{\mathtt{W}}'/\overline{\mathtt{Y}}][\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{V}_0'\mathtt{:}\overline{\mathtt{V}}'$
Since $\overline{\mathtt{Y}}$ do not appear in $\mathtt{V}_0'$ or $\overline{\mathtt{V}}'$, $\mathtt{S}_0\mathtt{:}\overline{\mathtt{S}}=[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\mathtt{V}_0'\mathtt{:}\overline{\mathtt{V}}'$.
And we then have $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{X}}]\vdash R_{p_3}\sqsubseteq_R R_{f_1}$, which by DS-Λ2 means Λ_3 and Λ_1 are disjoint.

□

Lemma 3 (Term Substitution Preserves Typing)**.** *If* $\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}\in\mathtt{T}$, $\Delta;\Lambda;\Gamma\vdash\overline{\mathtt{d}}\in\overline{\mathtt{S}}$, $\Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$*, then* $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}\in\mathtt{T}$.

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: Easy.
Case T-FIELD: By induction hypothesis, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{S}_0$, $\Delta\vdash\mathtt{S}_0<:\mathtt{T}_0$.
By Lemma 10, $\Delta\vdash fields(bound_\Delta(\mathtt{S}_0))=\overline{\mathtt{S}}\ \overline{\mathtt{g}}$, $\mathtt{f}_i=\mathtt{g}_i$, $\mathtt{T}_i=\mathtt{S}_i$.
By T-FIELD, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\mathtt{.f}_i\in\mathtt{T}_i$.
Case T-INVK: $\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{T}$
$\Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\mathtt{e}_0\in\mathtt{T}_0\qquad \Delta;\Lambda;\Gamma,\overline{\mathtt{x}}\mapsto\overline{\mathtt{T}}\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}}$
$\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{T}_0)=\overline{\mathtt{T}}\rightarrow\mathtt{T}\qquad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$
By induction hypothesis,
$\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\in\mathtt{T}_0\qquad \Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{T}}$
Clearly, $\Delta;\Lambda\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\mathtt{e}_0\mathtt{.n(}\overline{\mathtt{e}}\mathtt{)}\in\mathtt{T}$
Case T-NEW:
By induction hypothesis, $\Delta;\Lambda;\Gamma\vdash[\overline{\mathtt{d}}/\overline{\mathtt{x}}]\overline{\mathtt{e}}\in\overline{\mathtt{S}}'$, $\Delta\vdash\overline{\mathtt{S}}'<:\overline{\mathtt{S}}$.
By S-TRANS 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 4 (Weakening)**.** *Suppose* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\overline{\mathtt{N}}$ *ok and* $\Delta\vdash\mathtt{U}$ *ok.*

- *If* $\Delta\vdash\mathtt{S}<:\mathtt{T}$*, then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{S}<:\mathtt{T}$.
- *If* $\Delta\vdash\mathtt{S}$ *ok, then* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}}\vdash\mathtt{S}$ *ok.*
- *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* $\Delta,\overline{\mathtt{X}}<:\overline{\mathtt{N}};\Lambda;\Gamma\vdash\mathtt{e}\in\mathtt{T}$.

Proof. Easy by induction on the derivation of subtyping and expression typing rules. □

Lemma 5 (Type Substitution Preserves Subtyping)**.** *If* $\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash S<:T$*, and* $\Delta_1\vdash\overline{U}<:[\overline{U}/\overline{X}]\overline{N}$ *with* $\Delta_1\vdash\overline{U}$ *ok and none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash[\overline{U}/\overline{X}]S<:[\overline{U}/\overline{X}]T$*.*

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

Case S-REFL: Trivial
Case S-VAR:
If $X\notin\overline{X}$, then it is trivial.
If $X\in\overline{X}$, $bound_\Delta(X_i)=N_i$.
$[\overline{U}/\overline{X}]X_i=U_i$ $\quad[\overline{U}/\overline{X}]bound_\Delta(X_i)=[\overline{U}/\overline{X}]N_i$.
By assumption, $\Delta\vdash U_i<:[\overline{U}/\overline{X}]N_i$.
Case S-TRANS: By induction hypothesis.
Case S-CLASS: Trivial.

□

Lemma 6 (Pattern-matching Type Substitution Preserves Subtyping)**.** *If* $\Delta\vdash S<:T$*,* $\Delta;[\overline{W}/\overline{Y}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$*, and none of* $\overline{Y}$ *appears in* Δ*, then* $\Delta\vdash[\overline{W}/\overline{Y}]S<:[\overline{W}/\overline{Y}]T$*.*

Proof. By induction on the derivation of subtyping relation:

Case S-REFL: trivial.
Case S-VAR: If $X\notin\overline{Y}$, then it is trivial. If $X=Y_i$, then $[\overline{W}/\overline{Y}]X=W_i$, and by Lemma 17, $[\overline{W}/\overline{Y}]\Delta\vdash W_i<:[\overline{W}/\overline{Y}]P_i$.
Case S-TRANS, S-CLASS: easy.

□

Lemma 7 (Type Substitution Preserves Typing)**.** *If* $\Delta_1,\overline{X}<:\overline{N},\Delta_2;\Lambda;\Gamma\vdash e\in T$ *and* $\Delta_1\vdash\overline{U}<:[\overline{U}/\overline{X}]\overline{N}$ *where* $\Delta_1\vdash\overline{U}$ *ok, and none of* $\overline{X}$ *appears in* Δ_1*, then* $\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Lambda;[\overline{U}/\overline{X}]\Gamma\vdash[\overline{U}/\overline{X}]e\in[\overline{U}/\overline{X}]T$*.*

Proof. By induction on the derivation of $\Delta_1,\overline{X}<:\overline{N},\Delta_2;\Lambda;\Gamma\vdash e\in T$

Case T-VAR: Trivial
Case T-FIELD: Let $\Delta_3=\Delta_1,[\overline{U}/\overline{X}]\Delta_2$
$\Delta_1,\overline{X}<:\overline{N},\Delta_2;\Lambda;\Gamma\vdash e_0\in T_0\quad\Delta\vdash fields(bound_\Delta(T_0))=\overline{T}\,\overline{f}$
By induction hypothesis,
$\Delta_3;[\overline{U}/\overline{X}]\Lambda;[\overline{U}/\overline{X}]\Gamma\vdash[\overline{U}/\overline{X}]e_0\in[\overline{U}/\overline{X}]T_0$
By Lemma 9,
$\Delta_3\vdash bound_{\Delta_3}([\overline{U}/\overline{X}]T_0)<:[\overline{U}/\overline{X}]bound_{\Delta_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}}(T_0)$
By Lemma 10, $\Delta_3\vdash fields(bound_{\Delta_3}(S_0))=\overline{S}\ \overline{g}$, $f_j=g_j$, $S_j=[\overline{U}/\overline{X}]T_j$ for $j\le\#(\overline{f})$.
By T-FIELD, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2,[\overline{U}/\overline{X}]\Lambda;[\overline{U}/\overline{X}]\Gamma\vdash e_0.f_i\in[\overline{U}/\overline{X}]T_i$.
Case T-INVK: $\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash e_0.n(\overline{e})\in T$
$\Delta_1,\overline{X}<:\overline{N},\Delta_2;\Lambda;\Gamma\vdash e_0\in T_0\quad\Delta_1,\overline{X}<:\overline{N},\Delta_2;\Lambda;\Gamma\vdash\overline{e}\in\overline{S}$
$\Delta_1,\overline{X}<:\overline{N},\Delta_2;\Lambda\vdash mtype(n,T_0)=\overline{T}\rightarrow T$
$\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash\overline{S}<:\overline{T}$
By induction hypothesis,
$\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Lambda;[\overline{U}/\overline{X}]\Gamma\vdash[\overline{U}/\overline{X}]e_0\in[\overline{U}/\overline{X}]T_0$
$\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Lambda;[\overline{U}/\overline{X}]\Gamma\vdash[\overline{U}/\overline{X}]\overline{e}\in[\overline{U}/\overline{X}]\overline{S}$
By Lemma 5, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash[\overline{U}/\overline{X}]\overline{S}<:[\overline{U}/\overline{X}]\overline{T}$
By T-INVK,
$\Delta_1,[\overline{U}/\overline{X}]\Delta_2;[\overline{U}/\overline{X}]\Lambda;[\overline{U}/\overline{X}]\Gamma\vdash[\overline{U}/\overline{X}](e_0.n(\overline{e}))\in[\overline{U}/\overline{X}]T$
Case T-NEW: $\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash$ `new C<`$\overline{T}$`>`$(\overline{e})\in$ `C<`$\overline{T}$`>`
$\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash$ `C<`$\overline{T}$`>` *ok* $\quad\Delta\vdash fields($`C<`$\overline{T}$`>`$)=\overline{U}\,\overline{f}$
$\Delta_1,\overline{X}<:\overline{N},\Delta_2;\Lambda;\Gamma\vdash\overline{e}\in\overline{S}\quad\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash\overline{S}<:\overline{U}$
By Lemma 8, $\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash[\overline{U}/\overline{X}]$`C<`$\overline{T}$`>` *ok*
By Lemma 10,5 and induction hypothesis, conclusion follows.

□

Lemma 8. *If* $\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash T$ *ok, and* $\Delta_1\vdash\overline{U}<:[\overline{U}/\overline{X}]\overline{N}$ *with* $\Delta_1\vdash\overline{U}$ *okand none of* $\overline{X}$ *appearing in* Δ_1*, then* $\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash[\overline{U}/\overline{X}]T$ *ok.*

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

Case WF-OBJECT: easy.
Case WF-VAR: If X not in $\overline{X}$, then it's easy. Otherwise, $X=X_i$, $[\overline{U}/\overline{X}]X_i=U_i$, by assumption, $\Delta_1\vdash U_i$ *ok*. And by Lemma 4, conclusion follows.
Case WF-CLASS: By induction hypothesis and Lemma 5.

□

Lemma 9. *Suppose* $\Delta_1,\overline{X}<:\overline{N},\Delta_2\vdash T\,ok$*, and* $\Delta_1\vdash\overline{U}<:[\overline{U}/\overline{X}]\overline{N}$ *with* $\Delta_1\vdash\overline{U}$ *ok, and none of* $\overline{X}$ *appears in* Δ_1*. Then,* $\Delta_1,[\overline{U}/\overline{X}]\Delta_2\vdash bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T)<:[\overline{U}/\overline{X}](bound_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}(T))$

Proof. If T is a non-variable type, then the conclusion is trivial.
If T is a variable type, but $T\notin\overline{X}$, then the conclusion is also trivial.
If $T\in\overline{X}$,
$bound_{\Delta_1,[\overline{U}/\overline{X}]\Delta_2}([\overline{U}/\overline{X}]T)=U_i$, $[\overline{U}/\overline{X}](bound_{\Delta_1,\overline{X}<:\overline{N},\Delta_2}(T))=[\overline{U}/\overline{X}]N_i$.
By assumption and Lemma 4, conclusion follows.

□

Lemma 10. *If* $\Delta\vdash S<:T$*, and* $\Delta\vdash fields(bound_\Delta(T))=\overline{T}\ \overline{f}$*, then* $\Delta\vdash fields(bound_\Delta(S))=\overline{S}\ \overline{g}$*, and* $S_i=T_i$*, and* $g_i=f_i$ *for all* $i\le\#(\overline{f})$*.*

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

Case S-REFL: Trivial.
Case S-VAR: $bound_\Delta(S)=bound_\Delta(T)$. Conclusion follows.
Case S-TRANS: By induction hypothesis.
Case S-CLASS: $\Delta\vdash$ `C<`$\overline{T}$`>`$<:[\overline{T}/\overline{X}]T$
`class C<`$\overline{X}$`◁`$\overline{N}$`>◁T {`$\overline{S}'\overline{f}$ `...}`
$\Delta\vdash fields(bound_\Delta([\overline{T}/\overline{X}]T))=\overline{T}\,\overline{f}$
$\Delta\vdash fields($`C<`$\overline{T}$`>`$)=\overline{T}\ \overline{f},\ [\overline{T}/\overline{X}]\overline{S}'\overline{f}$
Conclusion is obvious.

□

Lemma 11. *If* $\Delta;\Lambda_1;\Gamma\vdash e\in T$*,* $\Delta;[\overline{W}/\overline{Y}]\vdash\Lambda_2\sqsubseteq_\Lambda\Lambda_1$*,* $\overline{Y}$ *do not appear in* Δ*, then* $\Delta;\Lambda_2;[\overline{W}/\overline{Y}]\Gamma\vdash e\in[\overline{W}/\overline{Y}]T$*.*

Proof. By induction on the derivation of $\Delta;\Lambda_1;\Gamma\vdash e\in T$

Case T-VAR: Trivial.
Case T-FIELD: $\Delta;\Lambda_1;\Gamma\vdash e_0.f_i\in T_i$
By T-FIELD and induction hypothesis,
$\Delta;\Lambda_1;\Gamma\vdash e_0\in T_0\qquad\Delta;\Lambda_2;[\overline{W}/\overline{Y}]\Gamma\vdash e_0\in[\overline{W}/\overline{Y}]T_0$
$\Delta\vdash fields(bound_\Delta(T_0))=\overline{T}\,\overline{f}$
By the definition of *fields*, it is easy to show that: $\Delta\vdash fields(bound_\Delta([\overline{W}/\overline{Y}]T_0))=[\overline{W}/\overline{Y}]\overline{T}\,\overline{f}$.
By T-FIELD, the conclusion follows.
Case T-INVK: $\Delta;\Lambda_1;\Gamma\vdash e_0.n(\overline{e})\in T$
$\Delta;\Lambda_1;\Gamma\vdash e_0\in T_0\qquad\Delta;\Lambda_1;\Gamma\vdash\overline{e}\in\overline{S}$
$\Delta;\Lambda_1\vdash mtype(n,T_0)=\overline{T}\rightarrow T\quad\Delta\vdash\overline{S}<:\overline{T}$
By induction hypothesis,
$\Delta;\Lambda_2;\Gamma\vdash e_0\in[\overline{W}/\overline{Y}]T_0\quad\Delta;\Lambda_2;\Gamma\vdash\overline{e}\in[\overline{W}/\overline{Y}]\overline{S}$
By Lemma 12, $\Delta;\Lambda_2\vdash mtype(n,[\overline{W}/\overline{Y}]T_0)=[\overline{W}/\overline{Y}](\overline{T}\rightarrow T)$
By Lemma 6 $\Delta\vdash[\overline{W}/\overline{Y}]\overline{S}<:[\overline{W}/\overline{Y}]\overline{T}$.
The conclusion is obvious through T-INVK.
Case T-NEW: Easy by induction hypthesis and definition of *fields*.

□

Lemma 12. *Suppose* $\Delta;[\overline{W}/\overline{Y}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$*, and* $\overline{Y}$ *do not appear in* Δ*. If* $\Delta;\Lambda'\vdash mtype(n,T)=\overline{U}\rightarrow U_0$*, then* $\Delta;\Lambda\vdash mtype(n,[\overline{W}/\overline{Y}]T)=[\overline{W}/\overline{Y}]\overline{U}\rightarrow[\overline{W}/\overline{Y}]U_0$.

Proof. By induction on the derivation of *mtype*

$Case$ MT-VAR-S: $\Delta;\Lambda'\vdash mtype(\mathtt{n},\ \mathtt{X})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$

$\Delta;\Lambda'\vdash mtype(\mathtt{n},\ bound_\Delta(\mathtt{X}))=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$

By induction,

$\Delta;\Lambda\vdash mtype(\mathtt{n},\ [\overline{\mathtt{W}}/\overline{\mathtt{Y}}]bound_\Delta(\mathtt{X}))=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\rightarrow[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{U}_0$.

$Case$ MT-VAR-R: $\Lambda'=\langle R'_p, o'R'_f\rangle \quad R'_p=(\mathtt{X},\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)$

By SB-Λ, $\Lambda=\langle R_p, oR_f\rangle \quad R'_p=(\mathtt{X},\ \mathtt{<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{V}}\rightarrow\mathtt{V}_0)$

By MT-VAR-R: $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{X})=\overline{\mathtt{V}}\rightarrow\mathtt{V}_0$.

By SB-Λ, SB-R, and UNI, $\overline{\mathtt{V}}=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$, $\mathtt{V}_0=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\ \mathtt{U}_0$

$Case$ MT-CLASS-S:

$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{N}\{\ldots\overline{\mathtt{M}}\}$

$\mathtt{U}_0\ \mathtt{m}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\ldots\}\in\overline{\mathtt{M}}$

By T-METH-S, none of $\overline{\mathtt{Y}}$ appear in $\mathtt{U}_0$ and $\overline{\mathtt{U}}$. Then $\mathtt{U}_0=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{U}_0$, $\overline{\mathtt{U}}=[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}$. Conclusion follows.

$Case$ MT-CLASS-R: $\mathtt{T}=\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$

$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{T}\{\ldots\overline{\mathfrak{M}}\}$

$\mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>for}(\mathbb{M}_p; o\mathbb{M}_f)\ \mathtt{S}_0\ *\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\ldots\}\in\overline{\mathfrak{M}}$

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

$R_p=(\mathtt{T}_i,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)) \quad R_p=(\mathtt{T}_j,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{V}}\rightarrow\mathtt{V}_0))$

$\Lambda_d=\langle R_p, oR_f\rangle$

$$\begin{cases} \Delta;\Lambda'\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r & \text{if } \mathtt{n}=\mathtt{m} \\ \Lambda_r=\Lambda' & \text{if } \mathtt{n}=* \end{cases}$$

$\Delta;[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]=\Lambda_r\sqsubseteq_\Lambda\Lambda_d$

$\Delta;\Lambda'\vdash mtype(\mathtt{n},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}](\overline{\mathtt{S}}\rightarrow\mathtt{S}_0)$

And $\overline{\mathtt{U}}=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\overline{\mathtt{S}}$, $\mathtt{U}_0=[\overline{\mathtt{T}}/\overline{\mathtt{X}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{S}_0$

Since $\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ *ok*, and $\overline{\mathtt{Y}}$ is not in the domain of Δ, $[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\mathtt{T}=\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$

If $\mathtt{n}=*$, by Lemma 18, $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}][\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda_d$. The conclusion follows.

If $\mathtt{n}=\mathtt{m}$, by Lemma 20, conclusion follows.

$Case$ MT-SUPER-R,MT-SUPER-S: By induction hypothesis.

□

Lemma 13. *Suppose* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$*, and* $\overline{\mathtt{Y}}$ *do not appear in* Δ*. If* $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$*, then* $\Delta;\Lambda'\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$*.*

Proof. By induction on the derivation of $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$

$Case$ MT-CLASS-S and MT-SUPER-S: obvious.

$Case$ MT-CLASS-R:

$CT(\mathtt{C})=\mathtt{class\ C<}\overline{\mathtt{X}}\triangleleft\overline{\mathtt{N}}\mathtt{>}\triangleleft\mathtt{T}\{\ \ldots\overline{\mathfrak{M}}\}$

$\mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{Q}}\mathtt{>for}(\mathbb{M}_p; o\mathbb{M}_f)\ \mathtt{S}_0\ *\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\ldots\}\ \in\overline{\mathfrak{M}}$

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

$R_p=(\mathtt{T}_i,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{<}\overline{\mathtt{Z}}\triangleleft\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{V}}\rightarrow\mathtt{V}_0) \quad R_f=(\mathtt{T}_j,[\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{V}}'\rightarrow\mathtt{V}'_0)$

$\Lambda_d=\langle R_p, oR_f\rangle \quad \Delta;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$

$\Delta;[\overline{\mathtt{O}}/\overline{\mathtt{Z}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$

By Lemma 20 $\Delta;\Lambda'\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$.

Conclusion follows.

$Case$ MT-CLASS-R: Similar to MT-CLASS-S, use Lemma 20.

□

Lemma 14. *If* $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{T})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$, $\Delta\vdash\mathtt{S}<:\mathtt{T}$*, where* $\mathtt{S}$ *is not a type variable, then* $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{S})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$*.*

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

$Case$ S-REFL: Trivial.

$Case$ S-VAR: Doesn't apply.

$Case$ S-TRANS: Easy by induction hypothesis.

$Case$ S-CLASS: $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{T})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$.

If $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})$ is retrieved using MT-SUPER-S or MT-SUPER-R, then the conclusion is obvious.

If $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{V}}\rightarrow\mathtt{V}$ by MT-CLASS-S, by T-METH-S and *override*, conclusion is obvious.

If $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})=\overline{\mathtt{V}}\rightarrow\mathtt{V}$ by MT-CLASS-R, again by T-METH-R and *override*, conclusion is obvious.

□

Lemma 15. *If* $\Delta;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$ *for some* $\mathtt{m}$ *and* Λ_d*, and* $\Delta;\Lambda_r;\Gamma\vdash\mathtt{e}\in\mathtt{T}$*. Then* $\Delta;\Lambda;\Gamma\vdash[\mathtt{m}/*]\mathtt{e}\in\mathtt{T}$*.*

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

$Case$ T-VAR: Trivial.

$Case$ T-FIELD: $\Delta;\Lambda_r;\Gamma\vdash\mathtt{e}_0\mathtt{.f}_i\in\mathtt{T}_i$.

$\Delta;\Lambda_r;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0\ \ \Delta\vdash fields(bound_\Delta(\mathtt{T}_0))=\overline{\mathtt{T}}\ \overline{\mathtt{f}}$

By induction hypothesis, $\Delta;\Lambda_d;\Gamma\vdash[\mathtt{m}/*]\mathtt{e}_0\in\mathtt{T}_0$

Conclusion follows from T-FIELD.

$Case$ T-INVK: $\Delta;\Lambda_r;\Gamma\vdash\mathtt{e}_0\mathtt{.n}(\overline{\mathtt{e}})\in\mathtt{T}$

$\Delta;\Lambda_r;\Gamma\vdash\mathtt{e}_0\in\mathtt{T}_0 \quad \Delta;\Lambda_r\vdash mtype(\mathtt{n},\ \mathtt{T}_0)=\overline{\mathtt{T}}\rightarrow\mathtt{T}$

$\Delta;\Lambda_r;\Gamma\vdash\overline{\mathtt{e}}\in\overline{\mathtt{S}} \quad \Delta\vdash\overline{\mathtt{S}}<:\overline{\mathtt{T}}$

By induction hypothesis,

$\Delta;\Lambda;\Gamma\vdash[\mathtt{m}/*]\mathtt{e}_0\in\mathtt{T}_0 \quad \Delta;\Lambda;\Gamma\vdash[\mathtt{m}/*]\overline{\mathtt{e}}\in\overline{\mathtt{S}}$

By Lemma 16, $\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{T}_0)=\overline{\mathtt{T}}\rightarrow\mathtt{T}$.

By T-INVK, the conclusion follows.

$Case$ T-NEW: Esay by induction hypothesis and T-NEW.

□

Lemma 16. *Suppose* $\Delta;\Lambda\vdash specialize(\mathtt{m},\ \Lambda_d)=\Lambda_r$ *for some* $\mathtt{m}$ *and* Λ_d*, if* $\Delta;\Lambda_r\vdash mtype(\mathtt{n},\ \mathtt{T})=\overline{\mathtt{S}}\rightarrow\mathtt{S}$*, then* $\Delta;\Lambda\vdash mtype([\mathtt{m}/*]\mathtt{n},\ \mathtt{T})=\overline{\mathtt{S}}\rightarrow\mathtt{S}$*.*

Proof. By induction on the derivation of $\Delta;\Lambda_r\vdash mtype(*,\ \mathtt{T})=\overline{\mathtt{S}}\rightarrow\mathtt{S}$:

$Case$ MT-VAR-R:

$\Lambda_r=\langle R_p, R_f\rangle \quad R_p=(\mathtt{T},\ \overline{\mathtt{S}}\rightarrow\mathtt{S})$

By the definition of *specialize*: $\Delta;\Lambda\vdash mtype(\mathtt{m},\ \mathtt{T})=\overline{\mathtt{S}}\rightarrow\mathtt{S}$.

$Case$ MT-VAR-S: Easy by induction hypothesis.

$Case$ MT-CLASS-S, MT-SUPER-S: Easy, since no conditions involve Λ_r.

$Case$ MT-CLASS-R: $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_r\sqsubseteq_\Lambda\Lambda_d$

If $\mathtt{n}=*$, the conclusion is obvious, since $\Delta;\Lambda\vdash specialize(\mathtt{m},\ \mathtt{T})=\Lambda_r$.

If $\mathtt{n}=\mathtt{m}$, inspecting the definition of *specialize* and using the induction hypothesis, we have:

$\Delta;\Lambda_r\vdash specialize(\mathtt{m},\ \mathtt{T})=\Delta;\Lambda\vdash specialize(\mathtt{m},\ \mathtt{T})$.

$Case$ MT-SUPER-R: Similar to MT-CLASS-R.

□

Lemma 17. *If* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda\sqsubseteq_\Lambda\Lambda'$*, and* $\overline{\mathtt{Y}}$ *do not appear anywhere in* Λ*,* $\Lambda'=\langle R_p, oR_f\rangle$*, where* $R_f=(\mathtt{T},\ \mathtt{<}\overline{\mathtt{Y}}\triangleleft\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}}\rightarrow\mathtt{U})$*, 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. By UNI, $\Delta,\overline{\mathtt{Y}}<:\overline{\mathtt{P}}\vdash\mathtt{W}_i\prec:_{\overline{\mathtt{Y}}}\mathtt{Y}_i$ for all $\mathtt{W}_i\in\overline{\mathtt{W}}$. We inspect the rules of PM-*:

The first rule that applies is PM-VAR: $bound_\Delta(\mathtt{W}_i)=\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$, $\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\prec:_{\overline{\mathtt{Y}}}[\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}/\mathtt{Y}_i]bound_\Delta(\mathtt{Y}_i)$, where $bound_\Delta(\mathtt{Y}_i)=\mathtt{P}_i$

We next prove by derivation of $\Delta\vdash\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\prec:_{\overline{\mathtt{Y}}}[\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}/\mathtt{Y}_i]\mathtt{P}_i$

$Case$ PM-REFL: trivial.

Case PM-CL: Let $\mathtt{P}_i{=}\mathtt{C<}\overline{\mathtt{S}}\mathtt{>}$. 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 $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}{=}\mathtt{C<}\overline{\mathtt{S}}\mathtt{>}$, and by S-REFL, conclusion follows.

Case PM-CL-S: Easy by induction hypothesis.

□

Lemma 18. *If* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda_1\sqsubseteq_\Lambda\Lambda_2$, $\Delta;[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash\Lambda_2\sqsubseteq_\Lambda\Lambda_3$, *then* $\Delta;[(\overline{\mathtt{W}}{:}\overline{\mathtt{Q}})/(\overline{\mathtt{Y}}{:}\overline{\mathtt{Z}})]\vdash\Lambda_1\sqsubseteq_\Lambda\Lambda_3$.

Proof. By Lemma 19 and the induction hypothesis. □

Lemma 19. *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,

$R_1{=}(\mathtt{T}_1,\ \mathtt{<}\overline{\mathtt{X}}{\lhd}\overline{\mathtt{N}}\mathtt{>}\overline{\mathtt{U}}{\rightarrow}\mathtt{U}_0)$ $\quad R_2{=}(\mathtt{T}_2,\ \mathtt{<}\overline{\mathtt{Y}}{\lhd}\overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{V}}{\rightarrow}\mathtt{V}_0)$

$\Delta\vdash\mathtt{T}_2{<:}\mathtt{T}_1$ $\quad \Delta,\overline{\mathtt{X}}{<:}\overline{\mathtt{N}},\overline{\mathtt{Y}}{<:}\overline{\mathtt{P}};[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash \mathit{unify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})$

$R_3{=}(\mathtt{T}_3,\ \mathtt{<}\overline{\mathtt{Z}}{\lhd}\overline{\mathtt{O}}\mathtt{>}\overline{\mathtt{V}}'{\rightarrow}\mathtt{V}'_0)$

$\Delta\vdash\mathtt{T}_3{<:}\mathtt{T}_2$ $\quad \Delta,\overline{\mathtt{Y}}{<:}\overline{\mathtt{P}},\overline{\mathtt{Z}}{<:}\overline{\mathtt{O}};[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\vdash \mathit{unify}(\mathtt{V}_0{:}\overline{\mathtt{V}},\ \mathtt{V}'_0{:}\overline{\mathtt{V}}')$

By S-TRANS, $\Delta\vdash\mathtt{T}_3{<:}\mathtt{T}_1$

By UNI,

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

$[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}_0{:}\overline{\mathtt{V}}{=}[\overline{\mathtt{Q}}/\overline{\mathtt{Z}}]\mathtt{V}'_0{:}\overline{\mathtt{V}}'$ $\quad$ 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 4, 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 20. *Suppose* $\Delta;[\overline{\mathtt{W}}/\overline{\mathtt{Y}}]\vdash\Lambda'\sqsubseteq_\Lambda\Lambda$. $\Delta;\Lambda\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d){=}\Lambda_r$ *if and only if* $\Delta;\Lambda'\vdash \mathit{specialize}(\mathtt{m},\ \Lambda_d){=}\Lambda'_r$, *and* $\Lambda_r{=}\Lambda'_r$.

Proof. In the *only if* direction:

$\Lambda_d{=}\langle R_p,oR_f\rangle$

$R_p{=}(\mathtt{T}_i,\ \mathtt{<}\overline{\mathtt{Z}}{\lhd}\overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}}{\rightarrow}\mathtt{U})$ $\quad R_f{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}{\rightarrow}\mathtt{V})$

$\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_i){=}\overline{\mathtt{U}}'{\rightarrow}\mathtt{U}'$ $\quad R'_p{=}(\mathtt{T}_i,\ \overline{\mathtt{U}}'{\rightarrow}\mathtt{U}')$

By Lemma 12,

$\Delta;\Lambda'\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_i){=}\overline{\mathtt{U}}'{\rightarrow}\mathtt{U}'$ $\quad R''_p{=}(\mathtt{T}_i,\ \overline{\mathtt{U}}'{\rightarrow}\mathtt{U}')$

Case If $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_j){=}\overline{\mathtt{V}}'{\rightarrow}\mathtt{V}'$, $R'_f{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}'{\rightarrow}\mathtt{V}')$
Then again, by Lemma 13,
$\Delta;\Lambda'\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_j){=}\overline{\mathtt{V}}'{\rightarrow}\mathtt{V}'$ $\quad R''_f{=}(\mathtt{T}_j,\ \overline{\mathtt{V}}'{\rightarrow}\mathtt{V}')$
The conclusion is then obvious.

Case If $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_j)$ is undefined, by Lemma 13, $\Delta;\Lambda'\vdash \mathit{mtype}(\mathtt{m},\ \mathtt{T}_j)$ must also be undefined. $R'_f{=}R''_f{=}R_f$.
The conclusion is then obvious.

In the *if* direction, the proof is exactly the same using Lemma 12 and 13.

□

Lemma 21 (Method Type Lookup Terminates)**.** *Method type lookup* $\mathit{mtype}(\mathtt{n},\ \mathtt{T})$ *for all* $\mathtt{T}$ *with a finite chain of reflective dependency either terminates with* $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T}){=}\overline{\mathtt{U}}{\rightarrow}\mathtt{U}_0$, *or ends with none of the MT-* rules applicable.*

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

$\mathit{refchain}(\mathtt{Object})$ $\quad = \mathtt{Object}$

$\mathit{refchain}(\mathtt{X})$ $\quad = \mathtt{X}{:}\mathit{refchain}(\mathit{bound}_\Delta(\mathtt{X}))$

$\mathit{refchain}(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>})$ $\quad = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}{:}\mathit{refchain}(\mathtt{T}_i){:}\mathit{refchain}([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})$

where $\overline{\mathtt{T}}{=}\mathtt{T}_0,\ldots,\mathtt{T}_n$

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:

$\mathit{measure}(\mathit{mtype}(\mathtt{n},\ \mathtt{T})){=}\mathit{length}(\mathit{refchain}(\mathtt{T}))$.

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

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

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

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

□