Morphing: Safely Shaping a Class in the Image of Others

Shan Shan Huang[1,2], David Zook[1], and Yannis Smaragdakis[2]

[1] Georgia Institute of Technology, College of Computing
{ssh,dzook}@cc.gatech.edu
[2] University of Oregon, Department of Computer and Information Sciences
yannis@cs.uoregon.edu

Abstract. We present MJ: a language for specifying general classes whose members are produced by iterating over members of other classes. We call this technique "class morphing" or just "morphing". Morphing extends the notion of genericity so that not only types of methods and fields, but also the *structure* of a class can vary according to type variables. This offers the ability to express common programming patterns in a highly generic way that is otherwise not supported by conventional techniques. For instance, morphing lets us write generic proxies (i.e., classes that can be parameterized with another class and export the same public methods as that class); default implementations (e.g., a generic do-nothing type, configurable for any interface); semantic extensions (e.g., specialized behavior for methods that declare a certain annotation); and more. MJ's hallmark feature is that, despite its emphasis on generality, it allows modular type checking: an MJ class can be checked independently of its uses. Thus, the possibility of supplying a type parameter that will lead to invalid code is detected early—an invaluable feature for highly general components that will be statically instantiated by other programmers.

1 Introduction

The holy grail of software construction is *separation of concerns*: aspects of program behavior should be treated independently, so that complexity can be decomposed into manageable pieces. Decomposition techniques have been the goal of programming languages for several decades, both with standard object-oriented techniques, as well as with "aspect" languages such as AspectJ [19] or JBoss AOP [6]. Nevertheless, all mechanisms offer a fundamental trade-off between generality and safety: if a mechanism is general, then it is hard to check that it is valid for all possible inputs. In this paper, we present a powerful modularity technique called *class morphing* or just *morphing*. We discuss morphing through MJ—a reference language that demonstrates what we consider the desired expressiveness and safety features of an advanced morphing language. MJ morphing can express highly general object-oriented components (i.e., generic classes) whose exact members are not known until the component is parameterized with concrete types. For a simple example, consider the following MJ class, implementing a standard "logging" extension:

```
class MethodLogger<class X> extends X {
  <Y*>[meth]for(public int meth (Y) : X.methods)
  int meth (Y a) {
    int i = super.meth(a);
    System.out.println("Returned: " + i);
    return i;
  }
}
```

MJ allows class `MethodLogger` to be declared as a subclass of its type parameter, `X`. The body of `MethodLogger` is defined by static iteration (using the `for` statement) over all methods of `X` that match the pattern `public int meth(Y)`. `Y` and `meth` are pattern variables, matching any type and method name, respectively. Additionally, the `*` symbol following the declaration of `Y` indicates that `Y` matches any number of types (including zero). That is, the above pattern matches all `public` methods that return `int`. The pattern variables are used in the declaration of `MethodLogger`'s methods: for each method of the type parameter `X`, `MethodLogger` declares a method with the same name and type signature. (This does not have to be the case, as shown later.) Thus, the exact methods of class `MethodLogger` are not determined until it is type-instantiated. For instance, `MethodLogger<java.io.File>` has methods `compareTo` and `hashCode`: these are the only `int`-returning methods of `java.io.File` and its superclasses.

"Reflective" program pattern matching and transformation, as in the above example, are not new. Several pattern matching languages have been proposed in prior literature (e.g., [2–4, 25]) and most of them specify transformations based on some intermediate program representation (e.g., abstract syntax trees) although the patterns resemble regular program syntax. Compared to such work, MJ is quite unique for two reasons:

– MJ makes reflective transformation functionality a natural extension of Java generics. For instance, our above example class `MethodLogger` appears to the programmer as a regular class, rather than as a separate kind of entity, such as a "transformation". Using a generic class is a matter of simple type-instantiation, which produces a regular Java class, such as `MethodLogger<java.io.File>`.
– MJ generic classes support modular type checking—a generic class is type-checked independently of its type-instantiations, and errors are detected if they can occur with *any* possible type parameter. This is an invaluable property for generic code: it prevents errors that only appear for some type parameters, which the author of the generic class may not have predicted. This problem has been the target of some prior work, such as type-safe reflection [10], compile-time reflection [11], and safe program generation [13]. Yet none of these mechanisms offer MJ's modular type checking guarantees. For instance, the Genoupe [10] approach has been shown unsafe, as the reasoning depends on properties that can change at runtime; CTR [11] only captures undefined variable and type incompatibility errors, does not offer a formal system or proof of soundness, and has limited expressiveness compared to MJ (especially with respect to method arguments); SafeGen [13] has no sound-

ness proof and relies on the capabilities of an automatic theorem prover—an unpredictable and unfriendly process from the programmer's perspective.

For an example of modular type checking, consider a "buggy" generic class:

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

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

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

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

2 Language Overview and Motivation

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

2.1 Language Basics

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

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

We overload the keyword `for` for static iteration. The iterator definition immediately follows `for`, delimited by parentheses. This defines the set of elements for iteration, which we call the *reflective range* (or just *range*) of the iterator. The iterator definition has the basic format *pattern* : *reflection set*. The *reflection set* is defined by applying the `.methods` or `.fields` keywords to a type, designating all methods or fields of that type. The *pattern* is either a method or field signature pattern, used to filter out elements from the reflection set. Only elements that match the pattern belong in the reflective range. In the example above, the reflective range contains only `static` methods of type `T`, with name `foo`, no argument, and return type `int`.

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

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

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

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

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

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

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

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

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

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

The above example shows a simple proxy class that declares methods that mimic the (non-`void`-returning) public methods of its type parameter. Declared method names are the original method names prefixed by the constant name `delegate`. Declared methods call the corresponding original methods after logging the call.

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

2.2 Applications

MJ opens the door for expressing a large number of useful idioms in a general, reusable way. This is the power of morphing features: we can shape a generic class or interface according to properties of the members of the type it is parameterized with. The morphing approach is similar to reflection, yet all reasoning is performed statically, there is syntax support for easily creating new fields and methods, and type safety is statically guaranteed.

Default Class. Consider a general "default implementation" class that adapts its contents to any interface used as a type parameter. The class implements all methods in the interface, with each method implementation returning a default value. This functionality is particularly useful for testing purposes—e.g., in the context of an application framework (where parts of the hierarchy will be implemented only by the end user), in uses of the Strategy pattern [12] with "neutral" strategies, etc. (Note that keyword `throws` in the pattern does not prevent methods with no exceptions from being matched, since `E` is declared to match a possibly-zero length vector of types.)

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

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

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

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

One can easily think of ways to enrich the above example with more complex default behavior, e.g., returning random values or calling constructor methods, instead of using statically determined default values. The essence of the technique, however, is in the iteration over existing methods and special handling of each case of return type. This is only possible because of MJ's morphing capabilities. In practice, random testing systems often implement very similar functionality (e.g., [8]) using unsafe run-time reflection. Errors in the reflective or code generating logic are thus not caught until they are triggered by the right combination of inputs, unlike in the MJ case.

Sort-by. A common scenario in data structure libraries is that of supporting sorting according to multiple fields of a type. Although one can use a generic sorting routine that accepts a comparison function, the comparison function needs to be custom-written for each field of a type that we are interested in. Instead, a simpler solution is to morph comparison functions based on the fields of a type. Consider the following implementation of an `ArrayList`, modeled after the `ArrayList` class in the Java Collections Framework:

```
public class ArrayList<E> extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
  ...// ArrayList fields and methods.

  // For each Comparable field of E, declare a sortBy method
  <F extends Comparable<F>>[f]for(public F f : E.fields)
  public void sortBy#f () {
    Collections.sort(this,
                     new Comparator<E> () {
                       public int compare(E e1, E e2) {
                         return e1.f.compareTo(e2.f);
                       }
                     });
  }
}
```

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

The examples above illustrate the power of MJ's morphing features. Yet more examples from the static reflection or generic aspects literature [10, 11, 13, 19] can be viewed as instances of morphing and can be expressed in MJ. For instance, the CTR work [11] allows the user to express a "transform" that iterates over methods of a class that have a `@UnitTestEntry` annotation and generate code to call all such methods while logging the unit test results. The same example can be expressed in MJ, with several advantages over CTR: MJ is better integrated in the language, using generic classes instead of a "transform" concept; MJ is a more expressive language, e.g., allowing matching methods with an arbitrary number and types of arguments; MJ offers much stronger guarantees of modular type safety, as its type system detects the possibility of conflicting definitions

(CTR only concentrates on preventing references to undefined entities) and we offer a proof of type soundness.

3 Type System: A Casual Discussion

Higher variability always introduces complexity in type systems. For instance, polymorphic types require more sophisticated type systems than monomorphic types, because polymorphic types can reference type "variables", whose exact values are unknown at the definition site of the polymorphic code. In MJ, in addition to type variables, there are also *name* variables—declarations and references can use names reflectively retrieved from type variables. Thus, the exact values of these names are not known when writing a generic class. Yet, the author of the generic class needs to have some confidence that his/her code will work correctly with any parameterization in its intended domain. The job of 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. Pattern matching type and name variables present two challenges: 1) how do we determine that declarations made with name variables are unique, i.e., there are no naming conflicts, and 2) how do we determine that references always refer to declared members and are well-typed, when we know neither the exact names of the members referenced, or the exact names of the members declared. In this section, we present through examples the main problems and insights related to MJ's modular type checking.

3.1 Uniqueness of Declarations

Simple case: Consider a simple MJ class:

```
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. For example:

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

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

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

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

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

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

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

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

In the above example, the uniquely identifying part of `X`'s method is no longer the uniquely identifying part of `IllegalChange<X>`'s method: the argument type of `X`'s method is no longer part of the argument type of `IllegalChange<X>`'s method. `IllegalChange<Overloaded>` (using the `Overloaded` class defined above) will cause an error in the generated code.

Multiple Reflective Blocks: We have discussed how to determine uniqueness within one reflective block. When there are multiple reflective blocks in the same type declaration, we need to guarantee that the declarations in one block do not conflict with the declarations in another block. One way to accomplish this is to guarantee that the blocks have iterators that produce *disjoint declaration ranges*.

Recall that the reflective range of an iterator is the set of entities it iterates over. Accordingly, we define the *declaration range* of an iterator to be the set of declarations it produces. Two ranges are disjoint if they contain no common members. Consider the following MJ class with two reflective blocks whose declaration ranges are disjoint:

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

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

The first block's reflective range contains all methods of `X` that take one argument of type `String`. The second block's reflective range contains all methods of `X` that take one argument of type `Number`. Thus, no methods in the first range can possibly be in the second range, and vice versa. Just as in previous examples, the uniqueness of entities in the reflective ranges implies the uniqueness of entities in the declaration ranges (since these use the same $\langle$name, argument types$\rangle$ tuple). Once we have guaranteed that declarations are unique both within and across reflective blocks, we can guarantee that all declarations within `TwoBlocks<X>` are unique, no matter what `X` is.

When using type variables as components of other types, disjointness is often hard to establish. Consider the following example:

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

  <R>[m] for ( R m (X) : X.methods )
  R m (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 have disjoint iteration ranges, they do *not* have disjoint declaration ranges. One instantiation that would cause a static error is the following:

```
class Overloaded2 {
  int m1 ( List<Overloaded2> a ) { ... }
  int m1 ( Overloaded2 a ) { ... }
}
```

`ManipulationError<Overloaded2>` would contain two methods named `m1`, both taking argument `List<Overloaded2>`.

In general, we can guarantee the uniqueness of declarations across reflective blocks by proving either type signature or name uniqueness. A general way to establish the uniqueness of declarations is by using unique static prefixes on names. (For static prefixes to be uniquely identifying, they must not be prefixes of each other.) For instance, 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) { ... }

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

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

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

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

Just as in the case of multiple iterators, the main issue is establishing the disjointness of declaration ranges, with the regular methods acting as a constant declaration range. Again, the easiest way to guarantee disjointness is through static prefixes such that all declarations produced by the reflective iterator have names distinct from `foo`.

Proper Method Overriding and Mixins: Proper overriding means that a subtype should not declare a method with the same name and arguments as a method in a supertype, but a non-covariant return type. Ensuring proper method overriding is again a special case of declaration range disjointness.

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 [5, 22]. Since the supertype could potentially be any type, we have no way of knowing its declarations. For instance, the following class is unsafe and will trigger a type error, as there is no guarantee that the superclass does not already contain an incompatible method `foo`.

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

Static prefixes are similarly insufficient to guarantee that subtype methods do not conflict with supertype methods. As a result, any legal type extending its type parameter can contain *no* members other than reflective iterators over its supertype that declare overriding versions for (some subset of) the supertype's methods.

3.2 Validity of References

Another challenge of modular type checking for a morphing language is to ensure the validity of references. We use the term "validity" to refer to the property that a referenced entity has a definition, and its use is well-typed. The following example demonstrates the complexities in checking reference validity in MJ:

```
class Reference<X> {
  Declaration<X> dx;
  ... // code to set dx field
  <U*>[n] for( String n (U) : X.methods )
  void n (U a) { dx.n(a); }
}
class Declaration<Y> {
  <V,W*>[m] for( V m (W) : Y.methods )
  void m (W a) { ... }
}
```

We would like to check the validity of method invocation `dx.n(a)`. There are multiple unknowns in this invocation that make checking its validity difficult:

- `dx` has type `Declaration<X>`, which has reflectively declared methods. We don't know statically these methods' names, argument types, or return types.
- the name of the method being invoked, `n`, is a name variable, reflectively matched to the method names in `X`, which is a type variable. Again, we do not know what these names may be.
- the type of the argument, `a`, is another type variable, `U`.

The intuition behind the checking logic is that if for every method `n` in `X` that takes any argument types `U`, and returns `String` (i.e., for every method in the range of the reflective block in `Reference<X>`) there is a method in `Declaration<X>` with the same name, taking the same types of arguments, then this reference is valid. The key to solving this problem is determining range *subsumption*. A range R_1 subsumes another range R_2 if all the entities in R_2 are also in R_1. We have already seen reflective ranges of an iterator and a declaration. We can easily expand the concept of range to other syntactic entities, such as arbitrary names and types. The range of a pattern matching type variable consists of all the types it matches in a given reflective iterator. Non-pattern-matching types have ranges with one element (themselves). The range of a name variable consists of all the names it matches in a given reflective iterator.

To determine the validity of `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>`, and the range of `U`, the actual argument type, is subsumed by the range of the formal argument type for methods in `Declaration<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 all methods in `X`, regardless of return type. Thus, the latter range subsumes the former. This guarantees that `Declaration<X>` does have a method matching each `n`. Similarly, the range of `U` consists of the argument types of methods in `X` that return `String`. The range of the argument types of methods in `Declaration<X>` consists of the argument types of all methods in `X`. The latter range subsumes the former. Therefore, we conclude that the call `dx.n(a)` is well-typed.

Subsumption of ranges in the MJ type system is checked by unification of names and type variables in the reflective predicates, followed by checking of type bounds (i.e., the known supertypes of type variables) for compatibility. The next section formalizes this type checking approach more precisely.

4 Formalization

We formalize a core subset of MJ's features. This formalization (FMJ) is based on the FGJ [16] formalism, with differences (other than the simple addition of our extra environment, Λ) highlighted in gray. Figures in which all rules are new to our formalism (Figures 4,5) are not highlighted at all, for better readability.

4.1 Syntax

The syntax of FMJ is presented in Figure 1. We adopt many of the notational conventions of FGJ: `C`,`D` denote constant class names; `X`,`Y` 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. In addition, we use `u` or `v` to denote name variables, while `n` denotes either variable or non-variable names.

We use the shorthand $\overline{\texttt{T}}$ for a sequence of types $\texttt{T}_0,\texttt{T}_1,\ldots,\texttt{T}_n$, and $\overline{\texttt{x}}$ for a sequence of unique variables $\texttt{x}_0,\texttt{x}_1,\ldots,\texttt{x}_n$. We use : for sequence concatenation. For example, $\overline{\texttt{S}}\texttt{:}\overline{\texttt{T}}$ is a sequence that begins with $\overline{\texttt{S}}$, followed by $\overline{\texttt{T}}$. We use $\in$ to mean "is a member of a sequence" (in addition to set membership). Thus, $\texttt{T}\in\overline{\texttt{T}}$ means that `T` is in the sequence $\overline{\texttt{T}}$. We use _ or ... for values of no particular significance to a rule. We use $\triangleleft$ and $\uparrow$ as shorthands for the keywords `extends` and `return`, respectively. Note that all classes must declare a superclass, which can be `Object`.

The goal of our formalization is to show that a type system in which both declarations and references can be made by reflecting over an unknown type can be sound. To keep the formalism comprehensible and concentrate on the core question, we left out some of MJ's language features. Most notable of these features is the ability to add static prefixes to name variables. Leaving this feature out prevents us from formalizing the declaration of both static and reflective methods in the same class or through inheritance, and from formalizing reflective iteration over different type variables.[1] We also do not formalize non-variable types as reflective parameters. This is a far less interesting case than reflecting over type variables, since all types and names are statically known. The zero or more length type vectors `T*` are also not formalized, without loss of generality. These type vectors are a matching convenience. They are treated as single types where they are used. Thus, safety issues regarding declaration and reference using vector types are covered by regular, non-vector types. Additionally, our formalism only includes reflectively declared methods, not fields—type checking reflectively declared fields is a strict adaptation of the techniques for checking methods. Lastly, polymorphic methods are not formalized.

Just like in FGJ, a program in FMJ is an (`e`, CT) pair, where `e` is an FMJ expression, and CT is the class table. We place the same conditions on CT as

[1] We could formalize the declaration of static and reflective methods in the same class (or through inheritance), but it would only be well-formed if the reflective methods are defined using *constant* method names (instead of name variables), and the constant names are different from all statically declared method names. This is technically uninteresting, and we leave it out of our formalism for simplicity. The same is true for formalizing reflective iteration over different type variables.

FGJ does. Every class declaration `class C...` has an entry in CT; `Object` is not in CT. In addition, 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.)

$$\begin{array}{lll}
\texttt{T} & ::= & \texttt{X | N}\\
\texttt{N} & ::= & \texttt{C<}\overline{\texttt{T}}\texttt{>}\\
\texttt{CL} & ::= & \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{>}\lhd\ \texttt{T}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\texttt{M}}\}\\
 & | & \texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{>}\lhd\ \texttt{T}\ \{\overline{\texttt{T}}\ \overline{\texttt{f}};\ \overline{\mathfrak{M}}\}\\
\texttt{M} & ::= & \texttt{T m (}\overline{\texttt{T}}\ \overline{\texttt{x}}\texttt{)}\ \{\uparrow\texttt{e;}\}\\
\mathfrak{M} & ::= & \texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>[}\overline{\texttt{u}}\texttt{] for(}\mathbb{M}\texttt{:X.methods) U n (}\overline{\texttt{U}}\ \overline{\texttt{x}}\texttt{)}\ \{\uparrow\texttt{e;}\}\\
\mathbb{M} & ::= & \texttt{V n (}\overline{\texttt{V}}\texttt{)}\\
\texttt{e} & ::= & \texttt{x | e.f | e.n(}\overline{\texttt{e}}\texttt{) | new C<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{) | (T)e}
\end{array}$$

Fig. 1. Syntax

4.2 Typing Judgments

The main typing rules of FMJ are presented in Figure 2, with auxiliary definitions presented in Figure 3, 4, 5, and 6. The core of this type system is in determining range subsumption and disjointness. Thus, we begin our discussion with an overview of the general typing rules, and follow with a detailed explanation of *subsumes* and *disjoint*, both defined in Figure 4.

There are three environments in our typing judgments:

– Δ: Type environment. Δ maps type variables to their upper bounds. Type variables can be introduced by class declarations (e.g., $\texttt{class C<}\overline{\texttt{X}}\lhd\overline{\texttt{N}}\texttt{>}$ ... introduces type variables $\overline{\texttt{X}}$), or by reflective iterator definitions (e.g., $\texttt{<}\overline{\texttt{Y}}\lhd\overline{\texttt{P}}\texttt{>[}\overline{\texttt{u}}\texttt{] for(...)}$ introduces type variables $\overline{\texttt{Y}}$).
– Γ: Variable environment. Γ maps variables (e.g., `x`) to their types.
– Λ: Reflective iteration environment. Λ is introduced with each reflective block. Λ maps a type `T` to a tuple of $\langle\overline{\texttt{Y}}, \overline{\texttt{u}}, \mathbb{M}\rangle$. `T` is the reflective parameter whose methods form the reflective set. $\mathbb{M}$ is the pattern used to filter the reflective set. $\overline{\texttt{Y}}$ and $\overline{\texttt{u}}$ are the pattern matching type and name variables introduced for use in $\mathbb{M}$ and the body of the reflective block. Since our syntax does not allow nested reflective loops, Λ contains at most one mapping.

A fourth environment, M, is sometimes used in the auxiliary definitions. M maps pattern matching type variables (e.g., those introduced by a reflective block) to other types, which may be pattern matching type variables, or non-pattern-matching types.

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

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

Uniqueness of Names: One of the main challenges of this type system is guaranteeing the uniqueness of declaration names. The uniqueness guarantee is simpler in our formalism than discussed in Section 3, since, in FMJ, a class can declare either static or reflective methods, but not both. Thus, we do not have to consider the case when static and reflective names conflict. We do, however, have to make sure that reflectively declared names do not conflict with each other. Rules T-METH-R and T-CLASS-R place conditions on well-typed methods and classes to prevent such naming conflicts.

T-METH-R ensures that methods declared within one reflective block should not conflict with 1) each other, and 2) methods in the superclass (i.e., there is proper overriding). The first condition is partly guaranteed by our syntax: a reflectively declared method must have the same name as the name in the method pattern for its enclosing reflective block.[2] Since a well-formed class can only be instantiated with other well-formed classes (WF-CLASS), and all well-formed classes have uniquely declared method names, we can be sure that method names reflectively retrieved from any type parameter through the pattern are unique.

The second condition is enforced using *override* (Figure 3). *override*(`n`, `T`, $\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$) determines whether method `n`, defined in some *subclass* of `T` with type signature $\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$, 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.[3] 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 $\Delta \vdash \mathit{validRange}(\Lambda, \mathtt{T})$ (Figure 4).

T-CLASS-R ensures that the reflective blocks within a well-typed class do not have declarations that conflict with each other. There are two key conditions: 1) all reflective blocks have the same reflective parameter ($\mathtt{X}_k$), and 2) the ranges of reflective blocks are disjoint pairwise. Since all blocks reflect over the same reflective parameter, which itself has unique method names, and no blocks overlap in their reflective ranges, the names used across all blocks are unique, as well. T-CLASS-R relies on the definition of *disjoint* to handle much of its complexity.

[2] This is a slightly different requirement than what is necessary in the implementation. In the formalization, there is no method name overloading, hence the uniquely identifying part of a method consists of its name only.

[3] Again, this is a simplification inherited from the FGJ formalism. In practice, one can overload method names with different argument types. We also made an extra 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.

Expression typing:

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

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

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

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

$$\frac{\begin{array}{c}\Delta;\Gamma;\Lambda \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ \mathit{ok} \\ \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}_0) <: \mathit{bound}_\Delta(\mathtt{T}) \ \text{ or } \ \Delta \vdash \mathit{bound}_\Delta(\mathtt{T}) <: \mathit{bound}_\Delta(\mathtt{T}_0)\end{array}}{\Delta;\Gamma;\Lambda \vdash (\mathtt{T})\,\mathtt{e}_0 \in \mathtt{T}} \qquad \text{(T-CAST)}$$

$$\frac{\begin{array}{c}\Delta;\Gamma;\Lambda \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}\ \mathit{ok} \\ \Delta \not\vdash \mathit{bound}_\Delta(\mathtt{T}_0) <: \mathit{bound}_\Delta(\mathtt{T}) \ \text{ and } \ \Delta \not\vdash \mathit{bound}_\Delta(\mathtt{T}) <: \mathit{bound}_\Delta(\mathtt{T}_0)\end{array}}{\Delta;\Gamma;\Lambda \vdash (\mathtt{T})\,\mathtt{e}_0 \in \mathtt{T}} \qquad \text{(T-SCAST)}$$

Method typing:

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

$$\frac{\begin{array}{c}\Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N}}, \overline{\mathtt{Y}} <: \overline{\mathtt{P}} \qquad \Gamma = \overline{\mathtt{x}} \mapsto \overline{\mathtt{V}}, \mathtt{this} \mapsto \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \qquad \Lambda = \mathtt{X}_i \mapsto \langle \overline{\mathtt{Y}},\ \overline{\mathtt{u}},\ \mathtt{U}_0\ \mathtt{n}\ (\overline{\mathtt{U}}) \rangle \\ \mathtt{X}_i \in \overline{\mathtt{X}} \qquad \Delta \vdash \overline{\mathtt{P}}, \mathtt{U}_0, \overline{\mathtt{U}}, \mathtt{V}_0, \overline{\mathtt{V}}\ \mathit{ok} \qquad \Delta;\Gamma;\Lambda \vdash \mathtt{e} \in \mathtt{S}_0 \qquad \Delta \vdash \mathtt{S}_0 <: \mathtt{U}_0 \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{T}\ \{\ \ldots\ \} \qquad \Delta;\Lambda \vdash \mathit{override}(\mathtt{n},\ \mathtt{T},\ \overline{\mathtt{V}} \to \mathtt{V}_0)\end{array}}{\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>[}\overline{\mathtt{u}}\mathtt{]for(U}_0\ \mathtt{n}\ (\overline{\mathtt{U}})\mathtt{:X}_i\mathtt{.methods)}\ \mathtt{V}_0\ \mathtt{n}\ (\overline{\mathtt{V}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\ \mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}} \qquad \text{(T-METH-R)}$$

Class typing:

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

$$\frac{\begin{array}{ll} & \Delta = \overline{\mathtt{X}} <: \overline{\mathtt{N}} \qquad \Delta \vdash \overline{\mathtt{N}}, \mathtt{T}, \overline{\mathtt{T}}\ \mathit{ok} \qquad \overline{\mathfrak{M}}\ \mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \qquad \mathtt{X}_k \in \overline{\mathtt{X}} \\ \text{for all} & \mathfrak{M}_i, \mathfrak{M}_j \in \overline{\mathfrak{M}}, \\ & \mathfrak{M}_i = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>[}\overline{\mathtt{u}}\mathtt{]for(U}_0\ \mathtt{n}_i\ \ (\overline{\mathtt{U}})\mathtt{:\ X}_k\mathtt{.methods)}\ \ldots \\ & \mathfrak{M}_j = \mathtt{<}\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{Q}}\mathtt{>[}\overline{\mathtt{v}}\mathtt{]for(V}_0\ \mathtt{n}_j\ \ (\overline{\mathtt{V}})\mathtt{:\ X}_k\mathtt{.methods)}\ \ldots \\ & \Lambda_i = \mathtt{X}_k \mapsto \langle \overline{\mathtt{Y}},\ \overline{\mathtt{u}},\ \mathtt{U}_0\ \mathtt{n}_i\ \ (\overline{\mathtt{U}}) \rangle \quad \Lambda_j = \mathtt{X}_k \mapsto \langle \overline{\mathtt{Z}},\ \overline{\mathtt{v}},\ \mathtt{V}_0\ \mathtt{n}_j\ \ (\overline{\mathtt{V}}) \rangle \\ \text{implies} & \Delta, \overline{\mathtt{Y}} <: \overline{\mathtt{P}}, \overline{\mathtt{Z}} <: \overline{\mathtt{Q}} \vdash \mathit{disjoint}(\langle \Lambda_i, \mathtt{n}_i, \mathtt{X}_k \rangle,\ \langle \Lambda_j, \mathtt{n}_j, \mathtt{X}_k \rangle)\end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{T}\ \{\ \overline{\mathtt{T}}\ \overline{\mathtt{f}};\ \overline{\mathfrak{M}}\}\ \mathtt{OK}} \qquad \text{(T-CLASS-R)}$$

Well-formed types:

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

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

Fig. 2. Typing Rules

Method type lookup:

$$\frac{\Lambda(\mathtt{X})=\langle\overline{\mathtt{Y}},\ \overline{\mathtt{u}},\ \mathtt{U}_0\ \mathtt{n}\ (\overline{\mathtt{U}})\rangle}{\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{X})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0}\qquad\text{(MT-VAR-R)}$$

$$\frac{\begin{array}{c}\mathtt{X}\notin dom(\Lambda)\quad\text{or}\quad\Lambda(\mathtt{X})=\langle\overline{\mathtt{Y}},\ \overline{\mathtt{u}},\ \mathtt{V}_0\ \mathtt{n}'(\overline{\mathtt{V}})\rangle\\ \Delta;\Lambda\vdash mtype(\mathtt{n},\ bound_\Delta(\mathtt{X}))=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0\end{array}}{\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{X})=\overline{\mathtt{U}}\rightarrow\mathtt{U}_0}\qquad\text{(MT-VAR-S)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C})=\mathtt{class\ C}<\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}>\lhd\ \mathtt{T}\ \{\ldots\ \overline{\mathtt{M}}\}\\ \Lambda\vdash constn(\mathtt{n})\quad\text{implies}\quad\mathtt{U}_0\ \mathtt{n}\ (\overline{\mathtt{U}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\in\overline{\mathtt{M}}\\ \Lambda\nvdash constn(\mathtt{n})\quad\text{implies}\quad\Lambda(\mathtt{C}<\overline{\mathtt{T}}>)=\langle\overline{\mathtt{Y}},\ \overline{\mathtt{u}},\ \mathtt{U}_0\ \mathtt{n}\ (\overline{\mathtt{U}})\rangle\end{array}}{\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{C}<\overline{\mathtt{T}}>)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{U}}\rightarrow\mathtt{U}_0)}\qquad\text{(MT-CLASS-S)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C})=\mathtt{class\ C}<\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}>\lhd\mathtt{T}\ \{\ldots\ \overline{\mathfrak{M}}\}\\ <\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>[\overline{\mathtt{u}}']\mathtt{for}(\mathtt{U}_0\ \mathtt{n}'\ (\overline{\mathtt{U}}):\mathtt{X}_i.\mathtt{methods})\ \mathtt{S}_0\ \mathtt{n}'\ (\overline{\mathtt{S}}\ \overline{\mathtt{x}})\ \{\uparrow\mathtt{e};\}\in\overline{\mathfrak{M}}\\ \Delta'=\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{Y}}<:\overline{\mathtt{P}})\qquad\Lambda'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{X}_i\mapsto\langle\overline{\mathtt{Y}},\ \overline{\mathtt{u}}',\ \mathtt{U}_0\ \mathtt{n}'\ (\overline{\mathtt{U}})\rangle)\\ \Delta';M\vdash subsumes(\langle\Lambda',\mathtt{n}',\mathtt{T}_i\rangle,\ \langle\Lambda,\mathtt{n},\mathtt{T}_i\rangle)\end{array}}{\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{C}<\overline{\mathtt{T}}>)=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](maptype_M(\overline{\mathtt{S}})\rightarrow maptype_M(\mathtt{S}_0))}\qquad\text{(MT-CLASS-R)}$$

$$\frac{\begin{array}{c}CT(\mathtt{C})=\mathtt{class\ C}<\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}>\lhd\ \mathtt{T}\ \{\ldots\ \overline{\mathtt{M}}\}\\ (\Lambda\vdash constn(\mathtt{n})\quad\mathtt{n}\notin\overline{\mathtt{M}})\quad\text{or}\quad(\Lambda\nvdash constn(\mathtt{n})\quad\mathtt{C}<\overline{\mathtt{T}}>\notin dom(\Lambda))\end{array}}{\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{C}<\overline{\mathtt{T}}>)=mtype(\mathtt{n},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T}>)}\qquad\text{(MT-SUPER-S)}$$

$$\frac{\begin{array}{ll} & CT(\mathtt{C})=\mathtt{class\ C}<\overline{\mathtt{X}}\lhd\overline{\mathtt{N}}>\lhd\mathtt{T}\ \{\ldots\ \overline{\mathfrak{M}}\}\\ \text{for all} & \mathfrak{M}\in\overline{\mathfrak{M}},\\ & \mathfrak{M}=<\overline{\mathtt{Y}}\lhd\overline{\mathtt{P}}>[\overline{\mathtt{u}}']\mathtt{for}(\mathtt{U}_0\ \mathtt{n}'\ (\overline{\mathtt{U}}):\mathtt{X}_i.\mathtt{methods})\ \ldots\\ & \Delta'=\Delta,[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\overline{\mathtt{Y}}<:\overline{\mathtt{P}})\quad\Lambda'=[\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{X}_i\mapsto\langle\overline{\mathtt{Y}},\ \overline{\mathtt{u}}',\ \mathtt{U}_0\ \mathtt{n}'\ (\overline{\mathtt{U}})\rangle)\\ \text{implies} & \Delta'\vdash disjoint(\langle\Lambda',\mathtt{n}',\mathtt{T}_i\rangle,\ \langle\Lambda,\mathtt{n},\mathtt{T}_i\rangle)\end{array}}{\Delta;\Lambda\vdash mtype(\mathtt{n},\ \mathtt{C}<\overline{\mathtt{T}}>)=mtype(\mathtt{n},\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{T})}\qquad\text{(MT-SUPER-R)}$$

Valid method overriding:

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

Field lookup:

$$fields(\mathtt{Object})=\bullet$$

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

Fig. 3. Method type lookup, overriding and field lookup.

Method range subsumption:

$$\frac{\begin{array}{c}\varDelta \vdash \mathtt{T}_1 <: \mathtt{T}_2 \qquad \overline{\mathtt{Y}} = \mathit{pmVars}(\varLambda_1) \\ \varDelta;\varLambda_1 \vdash \mathit{mtype}(\mathtt{n}_1,\ \mathtt{T}_1) = \overline{\mathtt{U}} \to \mathtt{U}_0 \qquad \varDelta;\varLambda_2 \vdash \mathit{mtype}(\mathtt{n}_2,\ \mathtt{T}_2) = \overline{\mathtt{V}} \to \mathtt{V}_0 \\ \varDelta;M;\overline{\mathtt{Y}} \vdash \mathit{tunify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}}) \qquad \langle \varLambda_2,\mathtt{n}_2 \rangle \sqsubseteq_{id} \langle \varLambda_1,\mathtt{n}_1 \rangle\end{array}}{\varDelta;M \vdash \mathit{subsumes}(\langle \varLambda_1,\mathtt{n}_1,\mathtt{T}_1 \rangle,\ \langle \varLambda_2,\mathtt{n}_2,\mathtt{T}_2 \rangle)}$$

Method range disjointness:

$$\frac{\varLambda_1 \vdash \mathit{constn}(\mathtt{n}_1) \qquad \varLambda_2 \vdash \mathit{constn}(\mathtt{n}_2) \qquad \mathtt{n}_1 \neq \mathtt{n}_2}{\varDelta \vdash \mathit{disjoint}(\langle \varLambda_1,\mathtt{n}_1,\mathtt{T}_1 \rangle,\ \langle \varLambda_2,\mathtt{n}_2,\mathtt{T}_2 \rangle)} \quad \text{(DS-NAME)}$$

$$\frac{\begin{array}{c}\varDelta;\varLambda_1 \vdash \mathit{mtype}(\mathtt{n}_1,\ \mathtt{T}_1) = \overline{\mathtt{U}} \to \mathtt{U}_0 \qquad \varDelta;\varLambda_2 \vdash \mathit{mtype}(\mathtt{n}_2,\ \mathtt{T}_2) = \overline{\mathtt{V}} \to \mathtt{V}_0 \\ \varLambda_1 \nvdash \mathit{constn}(\mathtt{n}_1) \text{ or } \varLambda_2 \nvdash \mathit{constn}(\mathtt{n}_2) \qquad \varDelta \vdash \mathtt{T}_1 <: \mathtt{T}_2 \text{ or } \varDelta \vdash \mathtt{T}_2 <: \mathtt{T}_1 \\ \overline{\mathtt{Y}} = \mathit{pmVars}(\varLambda_1) \quad \overline{\mathtt{Z}} = \mathit{pmVars}(\varLambda_2) \\ \text{for no } M, \quad \varDelta;M;\overline{\mathtt{Y}},\overline{\mathtt{Z}} \vdash \mathit{tunify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})\end{array}}{\varDelta \vdash \mathit{disjoint}(\langle \varLambda_1,\mathtt{n}_1,\mathtt{T}_1 \rangle,\ \langle \varLambda_2,\mathtt{n}_2,\mathtt{T}_2 \rangle)} \quad \text{(DS-TYPE)}$$

Subtype range validity:

$$\varDelta \vdash \mathit{validRange}(\emptyset,\ \mathtt{T}) \quad \text{(VR-NOREFL)} \qquad \varDelta \vdash \mathit{validRange}(\mathtt{X} \mapsto \langle \ldots \rangle,\ \mathtt{X}) \quad \text{(VR-VAR)}$$

$$\frac{\begin{array}{l} \qquad CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{S}\ \{\ \ldots\ \overline{\mathfrak{M}}\} \\ \qquad \varDelta \vdash \mathit{validRange}(\varLambda,\ [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{S}) \quad \varLambda = \mathtt{T} \mapsto \langle \_,\mathtt{n},\_ \rangle \\ \text{for all} \quad \mathfrak{M} \in \overline{\mathfrak{M}} \\ \qquad \mathfrak{M} = \mathtt{<}\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{Q}}\mathtt{>[}\overline{\mathtt{u}}'\mathtt{]\ for\ (S_0\ n'\ (}\overline{\mathtt{S}}\mathtt{)\ :\ X}_i\mathtt{.methods)\ \ldots} \\ \qquad \varDelta' = \varDelta,\overline{\mathtt{Z}} <: [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{Q}} \quad \varLambda' = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{X}_i \mapsto \langle \overline{\mathtt{Z}},\ \overline{\mathtt{u}}',\ \mathtt{S}_0\ \mathtt{n}'\ (\overline{\mathtt{S}}) \rangle) \\ \text{implies} \left\{ \begin{array}{l} \varDelta;M \vdash \mathit{subsumes}(\langle \varLambda',\mathtt{n}',\mathtt{T}_i \rangle,\ \langle \varLambda,\mathtt{n},\mathtt{T} \rangle) \text{ for some } M \quad \text{or} \\ \varDelta \vdash \mathit{disjoint}(\langle \varLambda',\mathtt{n}',\mathtt{T}_i \rangle,\ \langle \varLambda,\mathtt{n},\mathtt{T} \rangle) \end{array} \right. \end{array}}{\varDelta \vdash \mathit{validRange}(\varLambda,\ \mathtt{C<}\overline{\mathtt{T}}\mathtt{>})} \quad \text{(VR-CLASS)}$$

Identifier subrange rules:

$$\frac{\varLambda_2 \vdash \mathit{constn}(\mathtt{n}_2) \text{ implies } (\ \varLambda_1 \vdash \mathit{constn}(\mathtt{n}_1) \text{ and } \mathtt{n}_1 = \mathtt{n}_2\ )}{\langle \varLambda_1,\ \mathtt{n}_1 \rangle \sqsubseteq_{id} \langle \varLambda_2,\ \mathtt{n}_2 \rangle}$$

Constant name:

$$\frac{\varLambda = \mathtt{X} \mapsto \langle \_,\overline{\mathtt{u}},\_ \rangle \text{ implies } \mathtt{n} \notin \overline{\mathtt{u}}}{\varLambda \vdash \mathit{constn}(\mathtt{n})} \quad \text{(N-CONST)}$$

Pattern matching type variables of $\varLambda$:

$$\mathit{pmVars}(\emptyset) = \bullet \qquad \mathit{pmVars}(\mathtt{T} \mapsto \langle \overline{\mathtt{Y}},\ldots \rangle = \overline{\mathtt{Y}})$$

Type mapping application:

$$\frac{\mathtt{T} \notin \mathit{dom}(M)}{\mathit{maptype}_M(\mathtt{T}) = \mathtt{T}} \quad \text{(TM-VAR1)} \qquad \frac{M(\mathtt{X}) = \mathtt{T}}{\mathit{maptype}_M(\mathtt{X}) = \mathit{maptype}_M(\mathtt{T})} \quad \text{(TM-VAR2)}$$

$$\frac{\mathit{maptype}_M(\overline{\mathtt{T}}) = \overline{\mathtt{S}}}{\mathit{maptype}_M(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{C<}\overline{\mathtt{S}}\mathtt{>}} \quad \text{(TM-CLASS)}$$

Fig. 4. Reflection related auxiliary functions.

Type Unification:

$$\frac{M;\overline{\mathtt{Y}}\vdash \mathit{sunify}(\overline{\mathtt{T}},\ \overline{\mathtt{S}}) \quad \text{for all } \mathtt{Y}_i\in\overline{\mathtt{Y}},\ M(\mathtt{Y}_i)=\mathtt{T} \text{ implies } \Delta;\overline{\mathtt{Y}}\vdash \mathtt{T}\prec:\mathtt{Y}_i}{\Delta;M;\overline{\mathtt{Y}}\vdash \mathit{tunify}(\overline{\mathtt{T}},\ \overline{\mathtt{S}})}$$

Pattern matching rules:

$$\Delta;\overline{\mathtt{Y}}\vdash \mathtt{T}\prec:\mathtt{T} \qquad \text{(PM-EQ)}$$

$$\frac{\Delta;\overline{\mathtt{Y}}\vdash \overline{\mathtt{T}}\prec:\overline{\mathtt{S}}}{\Delta;\overline{\mathtt{Y}}\vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\prec:\mathtt{C<}\overline{\mathtt{S}}\mathtt{>}} \qquad \text{(PM-CLASS)}$$

$$\frac{\mathtt{Y}\in\overline{\mathtt{Y}} \quad \mathtt{T}\notin\overline{\mathtt{Y}} \quad \Delta;\overline{\mathtt{Y}}\vdash \mathtt{T}\prec:[\mathtt{T/Y}]\mathit{bound}_{\Delta}(\mathtt{Y})}{\Delta;\overline{\mathtt{Y}}\vdash \mathtt{T}\prec:\mathtt{Y}} \qquad \text{(PM-VAR1)}$$

$$\frac{\mathtt{Y}\in\overline{\mathtt{Y}} \quad \mathtt{T}\notin\overline{\mathtt{Y}} \quad \Delta\vdash \mathtt{T}<:\mathtt{T}' \quad \Delta;\overline{\mathtt{Y}}\vdash \mathtt{T}'\prec:\mathit{bound}_{\Delta}(\mathtt{Y})}{\Delta;\overline{\mathtt{Y}}\vdash \mathtt{T}\prec:\mathtt{Y}} \qquad \text{(PM-VAR2)}$$

$$\frac{\mathtt{Y}_1\in\overline{\mathtt{Y}} \quad \mathtt{Y}_2\in\overline{\mathtt{Y}} \quad \begin{cases} \Delta;\overline{\mathtt{Y}}\vdash [\mathtt{Y}_1/\mathtt{Y}_2]\mathit{bound}_{\Delta}(\mathtt{Y}_2)\prec:\mathtt{Y}_1 & \textit{or} \\ \Delta;\overline{\mathtt{Y}}\vdash [\mathtt{Y}_2/\mathtt{Y}_1]\mathit{bound}_{\Delta}(\mathtt{Y}_1)\prec:\mathtt{Y}_2 & \end{cases}}{\Delta;\overline{\mathtt{Y}}\vdash \mathtt{Y}_1\prec:\mathtt{Y}_2} \qquad \text{(PM-VAR3)}$$

Fig. 5. Type unification and pattern matching rules.

Valid Invocations: A second challenge in this type system is the validity of references to reflectively declared methods. T-INVK (Figure 2) specifies conditions for a well-typed method invocation. It uses $\Delta;\Lambda\vdash \mathit{mtype}(\mathtt{n},\ \mathtt{T})$ (Figure 3) to retrieve the type of method `n` in `T`, under the assumptions of Δ and Λ. We next highlight the *mtype* rules.

MT-VAR-R covers the case when we are looking for the type of method `n` in a type variable `X`, where `X` is the reflective parameter for the current reflective environment Λ. If the method pattern for the current reflective iterator uses `n` as its method name, *mtype*(`n`, `X`) is simply the type specified by the method pattern. MT-VAR-S covers the case when method `n` is *not* a method covered by the method pattern of the current reflective environment. In this case, we look for the type of `n` in the non-variable bound of `X`.

Rules MT-CLASS-S and MT-SUPER-S apply when we look for `n` in $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$, where $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$ has only statically declared methods. MT-CLASS-S states that if `n` is not a name variable used in the current reflective environment Λ (as determined by $\Lambda\vdash \mathit{constn}(\mathtt{n})$, Figure 4), and it is the name of a statically declared method in $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$, then *mtype* is defined to be the statically declared type of `n`, with proper type substitutions of $\overline{\mathtt{T}}$ for $\overline{\mathtt{X}}$. However, if `n` *is* a name variable in Λ, and $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ is the type that Λ iterates over, then the type of method covered by this name variable is exactly the type defined in the method pattern of Λ. MT-SUPER-S states that the type of `n` in $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ is the same as its type in $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$'s superclass, `T`, when `n` is a constant name, but not the name of a statically defined method in $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$, or when `n` is a name variable, but $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$ is not the type Λ iterates over.

Rules MT-CLASS-R and MT-SUPER-R apply when we look for `n` in $\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}$, where $\mathtt{C<}\overline{\mathtt{X}}\mathtt{>}$ has reflectively declared methods. As we explained in Section 3, the key in determining whether a reflectively declared method exists is in determin-

Subtyping rules:

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

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

Fig. 6. Subtyping rules.

ing that the range of `n` in the reference reflective environment is subsumed by the range of some name in the declaration reflective environment. If subsumption holds, the type of `n` is simply the type of the method whose name subsumes `n`, with the proper type substitutions of $[\overline{\mathtt{T}}/\overline{\mathtt{X}}]$, as well as the substitutions in mapping environment M. (The substitution for type `T` using M is defined as $\mathit{maptype}_M(\mathtt{T})$, in Figure 4. It is a straightforward application of type mappings.) MT-SUPER-R says that when the range of `n` in Λ is disjoint from every declared method range in $\mathtt{C<\overline{T}>}$, `n` has the same type as it does in $\mathtt{C<\overline{T}>}$'s superclass.

Subsumption: $\Delta;M\vdash \mathit{subsumes}(\langle \Lambda_1,\mathtt{n}_1,\mathtt{T}_1\rangle,\ \langle \Lambda_2,\mathtt{n}_2,\mathtt{T}_2\rangle)$, defined in Figure 4, determines whether, under the assumptions of Δ and M, the range of methods represented by $\mathtt{n}_1$ in type $\mathtt{T}_1$ subsumes the range of methods represented by $\mathtt{n}_2$ in type $\mathtt{T}_2$, under their respective reflective iteration environments.

There are three conditions for subsumption. First, $\mathtt{T}_1$ must be a subtype of $\mathtt{T}_2$. It only makes sense to compare ranges of methods if they are methods coming from the same class. Additionally, in defining reflective iterators, we interpret methods of a class (e.g., `X.methods`) to be the methods declared in the class and all of its superclasses, transitively. Thus, a subclass has more methods than its superclass, potentially yielding a larger range.

Secondly, the name $\mathtt{n}_1$ must be less strict than the name $\mathtt{n}_2$. Since all name variables can match any name, the only restriction on $\mathtt{n}_2$ is when $\mathtt{n}_1$ is a constant. In this case, $\mathtt{n}_2$ must be equal to $\mathtt{n}_1$ (see $\sqsubseteq_{id}$ in Figure 4).

Lastly, M must be a *one-way* unification mapping that maps the type signature of the larger range ($\overline{\mathtt{U}}\rightarrow\mathtt{U}_0$) onto that of the smaller range ($\overline{\mathtt{V}}\rightarrow\mathtt{V}_0$). This is a one-way unification because we want to ensure that one range is larger than the other and not just that their intersection is non-empty. *subsumes* uses $\Delta;M;\overline{\mathtt{Y}}\vdash \mathit{tunify}(\mathtt{U}_0{:}\overline{\mathtt{U}},\ \mathtt{V}_0{:}\overline{\mathtt{V}})$ to determine whether M is a proper unification mapping. We discuss *tunify* in detail shortly. But the main point to note is that its rules are quite general, and determine whether M is a *two-way* unification between its arguments, using the given pattern matching type variables. We use *tunify* to check one-way unification by using only $\overline{\mathtt{Y}}$ (the pattern matching type variables in the larger range) as the variables with respect to unification, ignoring the pattern matching type variables of Λ_2, which are considered constants.

Disjointness: *disjoint* (Figure 4) takes the same arguments as *subsumes*. The goal of *disjoint* is to determine the non-overlap of the *names* of the two method ranges. DS-NAME describes the easy case, when both ranges use constant names in their method patterns, but the names are not equal to each other. DS-TYPE describes disjointness conditions when at least one of the names is not a constant.

First, it stipulates that there can be no unification mapping between the types of the two method ranges. Here again, we use *tunify*. However, note that we pass the pattern matching type variables from both reflective ranges ($\overline{\mathtt{Y}}$ and $\overline{\mathtt{Z}}$) to *tunify*—we are looking for a two-way unification, in contrast to the one-way unification that *subsumes* looks for. Lack of unification between the two type signatures means that there is no method whose type signature is in both ranges. However, this is not enough to determine the disjunction of the names covered by these ranges—if the methods range over classes from completely different inheritance hierarchies, they could have disjoint method types, but still the same names. Thus, DS-TYPE requires that either $\mathtt{T}_1$ be a subtype of $\mathtt{T}_2$, or vice versa. If methods from the same inheritance hierarchy have different types, then they definitely have distinct names.

Unification: $\Delta;M;\overline{\mathtt{Y}}\vdash tunify(\overline{\mathtt{T}},\ \overline{\mathtt{S}})$, defined in Figure 5, determines whether $\overline{\mathtt{T}}$ and $\overline{\mathtt{S}}$ can be unified by unification mapping M, using $\overline{\mathtt{Y}}$ as the pattern matching type variables. Δ is the type environment under which $\overline{\mathtt{Y}}$, $\overline{\mathtt{T}}$, and $\overline{\mathtt{S}}$ are properly defined. *tunify* first checks that M is a proper *syntactic* unifying mapping. Syntactic unification is a common two-way unification such that, after the mapping is applied to $\overline{\mathtt{T}}$ and $\overline{\mathtt{S}}$, the resulting type sequences $\overline{\mathtt{T}}'$and $\overline{\mathtt{S}}'$are syntactically equivalent. The precise definition of *sunify* is elided for space reasons. Interested readers can obtain the specifics from the technical report [14]. What syntactic unification does not check, however, is whether a mapping from pattern matching type variable Y to type T conforms to the bound of Y in Δ. Thus, *tunify* uses the pattern matching relation, $\prec:$, to check that T can indeed be matched by Y.

The pattern matching relation, $\Delta;\overline{\mathtt{Y}}\vdash\mathtt{T}\prec:\mathtt{S}$ (Figure 5), holds if there exists a type that can be matched by both T and S. $\overline{\mathtt{Y}}$ are the pattern matching type variables, and Δ is the type environment under which all types are well-formed. The interesting case is in determining whether a non-pattern-matching type T can be matched by a pattern matching type Y. The intuition is that T can be matched by Y if it is within the bound of Y. This means that, with proper type substitutions, either T can be matched by the bound of Y (PM-VAR1), or T's superclass can be matched by the bound of Y (PM-VAR2). We use PM-VAR3 to determine whether there is a type that can be matched by two pattern matching type variables, $\mathtt{Y}_1$ and $\mathtt{Y}_2$. The intuition is that if there exists a type that can be matched by both $bound_{\Delta}(\mathtt{Y}_1)$ and $\mathtt{Y}_2$ (or $bound_{\Delta}(\mathtt{Y}_2)$ and $\mathtt{Y}_1$), then there is a type that can be matched by both $\mathtt{Y}_1$ and $\mathtt{Y}_2$.

4.3 Soundness

We prove soundness using the familiar subject reduction and progress theorems.

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

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

In addition, we must prove a lemma regarding the uniqueness of names—can there be multiple methods declared with the same name? A closer inspection of the MT-CLASS-R rule shows that there appears to be some non-determinism: the second condition of the rule specifies that one of the reflective blocks in $\overline{\mathfrak{M}}$ makes the conditions that follow true. We prove in the following lemma that there can *only* be one such $\mathfrak{M}$ in class `C`:

Lemma 1 [Name Uniqueness]: If $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ ok, $CT(\texttt{C})=\texttt{class C<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\triangleleft\texttt{T \{ ... }\overline{\mathfrak{M}}\}$, then there can be at most one $\mathfrak{M}_i\in\overline{\mathfrak{M}}$ such that $\Delta;\Lambda\vdash mtype(\texttt{n, C<}\overline{\texttt{T}}\texttt{>})=\overline{\texttt{U}}\rightarrow\texttt{U}_0$.

Full text of the proofs, reduction rules, and related functions are defined in the technical report version of this paper[14].

5 Discussion

Design Discussion. MJ can be viewed as part of a general effort to bring meta-programming constructs to mainstream programming languages, with smooth integration of features and modular type checking guarantees. In this sense, it is interesting to discuss MJ's design and implementation decisions in comparison with our other concurrent project: cJ [15]. cJ is an extension of Java with a static-if construct, allowing the configuration of generic classes based on properties of their type parameters. For instance, cJ can express a `List<X>` class that implements `Serializable` only when its type parameter `X` implements `Serializable`.

cJ adds to Java a reflective "if", whereas MJ adds a reflective "for", as well as the ability to create declarations with non-constant names. Thus, it should not be a surprise that MJ is a more ambitious language with significantly more complexity. This is reflected clearly in our design decisions. cJ is designed with backward compatibility in mind, enabling an erasure-based translation. cJ language constructs can be "erased" producing regular Java code in a one-to-one correspondence between cJ generic classes and Java generic classes. Additionally, cJ interacts smoothly with advanced features in the Java type system, such as variance [24, 17] and polymorphic methods. In contrast, MJ takes a more radical approach, favoring feature-richness and integration of ideas over backward compatibility and implementation integration. This difference is most evident in MJ's implementation, which employs an expansion-based translation. MJ generic classes produce one regular non-generic Java class per instantiation. This implementation approach is harder to support in conjunction with some of Java's features (e.g., dynamic loading) but yields more power—e.g., to express mixins as generic subclasses. Furthermore, we have not concerned ourselves with supporting features such as variance and polymorphic methods. Considering the interaction of these features with MJ is part of future work. Overall, we do not view MJ as a language extension that can be easily integrated in standard Java. (After all, integrating with standard Java seems a near-hopeless proposition even for more modest research proposals, as the Java language has matured and the rate of change has decreased dramatically.) Instead, we view MJ as a more radical idea, intended to demonstrate the principles of morphing and to influence future language designers. Our goal with MJ is to show the first morphing language with a sound modular type checking system, and a smooth integration of concepts in an object-oriented framework.

Contrast with Meta-Programming and AOP Tools. Generally, few language mechanisms allow expressing what MJ does: writing one piece of code and having it be applied to multiple methods with different signatures. In the past, this has been the hallmark property of Meta-Object Protocols [9, 18] and later Aspect-Oriented Programming [20]. Neither mechanism offers modular safety guarantees, however. The same capabilities can be achieved with traditional reflection and program generation but with lower-level means of syntax-matching and, again, no safety guarantees.

An interesting special case of program generation is *staging languages* such as MetaML [23] and MetaOCaml [7]. 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 meta-programming tools, such as Template Haskell [21] are explicitly not modularly type safe—its authors acknowledge that they sacrifice the MetaML guarantees for expressiveness.

6 Future Work and Conclusions

There are several interesting directions of further work on MJ. A major one is the introduction of anti-patterns in addition to patterns. Several modular type checking scenarios require not just matching all entities that satisfy a pattern, but also ensuring that no entity exists that matches a certain other pattern. Anti-patterns increase the expressiveness of a morphing language significantly. For instance, they expand the possibilities for generating methods and fields with guarantees that they will not conflict with existing members of a type. Our introduction of anti-patterns will be based on the same type checking insights as patterns, namely on checking of range disjointness and subsumption.

Overall, we consider MJ and the idea of morphing to be a significant step forward in the expressiveness of modern programming languages. Morphing can be viewed as an aspect-oriented technique, allowing the extension and adaptation of existing code components, and enabling a single enhancement to affect multiple code sites (e.g., all methods of a class, regardless of name). Yet morphing is also deeply different from aspect-oriented programming, and can perhaps be seen as a bridge between AOP and generic programming. Morphing does not introduce functionality to unsuspecting code. Instead, it ensures that any extension is under the full control of the programmer. The result of morphing is a new class or interface, which the programmer is free to integrate in the application at will. Morphing strives for smooth integration in the programming language, all the way down to modular type checking. Thus, reasoning about morphed classes is possible, unlike reasoning about and type checking of generic aspects, which can typically only be done after their application to a specific code base. We thus view morphing as an exciting new direction in programming language research and MJ as an excellent ambassador of the approach.

Acknowledgments

This work was supported by the National Science Foundation under Grant No. CCR-0238289. We thank the Eugene Running Company, Leon Trotsky, and Eugene area pubs for inspiration. Opinions presented in this paper do not necessarily reflect those of the NSF. Or of Leon Trotsky.

References

1. E. Allen, J. Bannet, and R. Cartwright. A first-class approach to genericity. In *Proc. of the 18th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, pages 96–114, Anaheim, CA, USA, 2003. ACM Press.
2. J. Bachrach and K. Playford. The Java syntactic extender (JSE). In *Proc. of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 31–42, Tampa Bay, FL, USA, 2001. ACM Press.
3. J. Baker and W. C. Hsieh. Maya: multiple-dispatch syntax extension in Java. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 270–281, Berlin, Germany, 2002. ACM Press.
4. D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: tools for implementing domain-specific languages. In *Proc. of the Fifth Intl. Conf. on Software Reuse*, pages 143–153, Victoria, BC, Canada, 1998. IEEE.
5. G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proc. of the European conference on object-oriented programming on Object Oriented Programming Systems, Languages, and Applications*, pages 303–311, Ottawa, Canada, 1990. ACM Press.
6. B. Burke et al. *JBoss AOP Web site, http://www.jboss.org/products/aop*. Accessed Apr. 2007.
7. C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using ASTs, gensym, and reflection. In *Proc. of the 2nd Intl. Conf. on Generative Programming and Component Engineering*, LNCS 2830, pages 57–76. Springer-Verlag, 2003.
8. C. Csallner and Y. Smaragdakis. JCrasher: An automatic robustness tester for Java. *Software—Practice and Experience*, 34(11):1025–1050, Sept. 2004.
9. S. Danforth and I. R. Forman. Reflections on metaclass programming in SOM. In *Proc. of the 9th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 440–452, New York, NY, USA, 1994. ACM Press.
10. D. Draheim, C. Lutteroth, and G. Weber. A type system for reflective program generators. In *Proc. of the 4th Intl. Conf. on Generative Programming and Component Engineering*, LNCS 3676, pages 327–341, Tallin, Estonia, 2005. Springer-Verlag.
11. M. Fähndrich, M. Carbin, and J. R. Larus. Reflective program generation with patterns. In *Proc. of the 5th Intl. conference on Generative Programming and Component Engineering*, pages 275–284, Portland, OR, USA, 2006. ACM Press.

12. E. Gamma, R. Helm, and R. Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995.
13. S. S. Huang, D. Zook, and Y. Smaragdakis. Statically safe program generation with SafeGen. In *Proc. of the 4th Intl. Conf. on Generative Programming and Component Engineering*, LNCS 3676, pages 309–326, Tallin, Estonia, 2005. Springer-Verlag.
14. S. S. Huang, D. Zook, and Y. Smaragdakis. Morphing: Safely shaping a class in the image of others. Technical report, 2006. `http://www.cc.gatech.edu/∼ssh/mjfull.pdf`.
15. S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with safe type conditions. In *Proc. of the 6th Intl. Conf. on Aspect-Oriented Software Development*, Vancouver, Canada, 2007. ACM Press.
16. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In L. Meissner, editor, *Proc. of the 14th ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications*, volume 34(10), pages 132–146, 1999.
17. A. Igarashi and M. Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Trans. Program. Lang. Syst.*, 28(5):795–847, 2006.
18. G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
19. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
20. G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. of the 11th European Conf. on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
21. T. Sheard and S. P. Jones. Template meta-programming for Haskell. In *Proc. of the ACM SIGPLAN workshop on Haskell*, pages 1–16, Pittsburgh, Pennsylvania, 2002. ACM Press.
22. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of the 12th European Conf. on Object-Oriented Programming*, pages 550–570. Springer-Verlag LNCS 1445, 1998.
23. W. Taha and T. Sheard. Multi-stage programming with explicit annotations. In *Proc. of the 1997 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation*, pages 203–217, Amsterdam, The Netherlands, 1997. ACM Press.
24. M. Torgersen, C. P. Hansen, E. Ernst, P. von der Ahe, G. Bracha, and N. Gafter. Adding wildcards to the java programming language. In *Proc. of the 2004 ACM Symposium on Applied Computing*, pages 1289–1296, Nicosia, Cyprus, 2004. ACM Press.
25. E. Visser. Program transformation with Stratego/XT: Rules, strategies, tools, and systems in Stratego/XT 0.9. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, pages 216–238. Springer-Verlag, 2004. LNCS 3016.