Implementing Fully Modular, Statically Typed, Symmetric Multimethod Dispatch

Guy L. Steele Jr.
Oracle Labs
guy.steele@oracle.com

David Chase
Oracle Labs
david.r.chase@oracle.com

Eric Allen
Oracle Labs
eric.allen@oracle.com

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

Sukyoung Ryu
KAIST
sryu@cs.kaist.ac.kr

Abstract

The Fortress programming language integrates traditional mathematical notation into an object-oriented framework based on traits with multiple inheritance, overloading (of both methods and functions) resolved by symmetric dynamic dispatch, static types, and separately compiled modules. One innovation is *functional methods*, which (like conventional "dotted methods") are declared within traits and may be inherited, but are invoked by ordinary function calls (or mathematical operator syntax) rather than conventional "dotted method calls," and therefore compete in overloading resolution with ordinary function declarations. A component/API system governs visibility of traits, objects, and functions, and allows separate compilation of components.

A longstanding problem with multiple inheritance is what to do when methods inherited from several parents conflict. Many approaches have been explored in the literature; most fail to obey the intuitively desirable requirement that the function or method invoked be the uniquely most specific one that is both accessible and applicable. Fortress requires that the signatures in every overload set form a meet-bounded lattice; therefore it is impossible for any function or method call to be ambiguous. This idea goes back nearly two decades, but Fortress appears to be the first programming language to adopt and statically enforce it. Because this rule guarantees confluence, it enables a distributed implementation of dispatching that allows selective export and selective optimization.

We exhibit a source-to-source rewrite from a source language (a stripped-down version of Fortress) to a related target language that is simpler than the Java™ programming language and is readily supported by the Java Virtual Machine. The demonstrated rewriting is a practical basis for separate compilation and is easily extended to explicitly type-parameterized methods and functions.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, inheritance, modules, packages, polymorphism

General Terms Languages

Keywords object-oriented programming, traits, multiple dispatch, symmetric dispatch, multiple inheritance, overloading, modularity, methods, multimethods, functional methods, static types, run-time types, ilks, components, separate compilation, Fortress, meet rule

1. Introduction and Background

A longstanding problem with multiple inheritance is what to do when methods inherited from several parents conflict. This is actually an important special case of the more general problem of what to do when a method or function is overloaded, that is, having multiple definitions that may arise from a variety of sources, of which inheritance from a parent is one possibility. Many approaches have been explored in the literature; some unfortunately violate the intuitively desirable requirement that, for any given invocation, the function or method definition invoked be the uniquely most specific one that is both accessible and applicable. The problem is that there might be two (or more) accessible definitions that are applicable, equally specific, and more specific than any of the others (in such a situation, we say that the invocation is *ambiguous*). Many language designs solve this problem by adopting some sort of asymmetric "tie-break" rule that arbitrarily chooses one definition in preference to others that are otherwise equally specific; for example, in the context of inheritance, one might examine the textual order in which the parents are declared and prefer definitions from parents that occur earlier (or later) in this ordering.

The Fortress programming language, on the other hand, addresses the problem by simply prohibiting it from arising: the signatures in every overload set are required to form a meet-bounded lattice (the "Meet Rule"), and therefore it is impossible for any function or method call to be ambiguous. This idea goes back nearly two decades to the work of Castagna *et al.* [6], but Fortress appears to be the first programming language to adopt and statically enforce it. Our experience is that the requirement feels natural in practice, and static enforcement by a compiler helps to expose programming errors. Moreover, enforcing the requirement has two major benefits: (a) Because the lattice structure guarantees a confluence property, there is a modular strategy for implementing dynamic multimethod dispatch in a simple, distributed fashion. (b) The distributed implementation of mul-

timethod dispatch in turn makes it easy to divide programs into components that can be separately compiled, with full static type checking and full static enforcement of the Meet Rule, and with fine-grained control over the export and import of not only type declarations but also individual function declarations and definitions.

A general issue with method inheritance is the "super problem": how to allow a method that overrides an inherited method definition to invoke that overridden method. In the Java™ programming language, a method definition (as opposed to an abstract declaration) can be inherited from at most one parent, so it suffices to use the keyword `super`:

```
class MyType extends MyFather {
  double myMethod(int a, double b) {
    return Math.sin(b) + super.myMethod(a, b);
  }
}
```

But in a language that allows definitions to be inherited from multiple parents, it is necessary to specify which definition is desired. Rather than, say, naming the parent from which the definition should be taken, Fortress uses the dispatch mechanism itself to select to desired definition by allowing the programmer to use the keyword `asif` to indicate that an expression should be assumed to have a specified supertype of its usual "run-time type" for dispatch purposes:

```
trait MyType extends { MyFather, MyMother }
  myMethod(a: ℤ32, b: ℝ64): ℝ64 =
    (sin(b) + (self asif MyMother).myMethod(a, b)
       −(self asif MyFather).myMethod(3, b))
end
```

There is one more piece to the story. The Fortress language is designed to support many aspects of conventional mathematical notation, and many mathematical operators such as $+$ and $\leq$ and $\cup$ and especially $\cdot$ and $\times$ have different definitions depending on the types of the arguments; in a language with multiple dispatch, it is natural to define these operators as overloaded functions. On the other hand, there is organizational value in arranging the mathematical types into an object-oriented hierarchy, and so it would seem even more natural to define operators as overloaded methods. This is not a new idea, and it leads to the so-called "binary method problem" [5]: how to define a method that takes an argument of the same type as the receiver, in such a way as to preserve subtype relationships.

Fortress provides a novel solution to the binary method problem: functional methods, which have two distinctive characteristics: (i) a functional method may designate any argument, not just the textually leftmost, to be treated as the "dispatch target" or "receiver"; (ii) functional methods are inherited like conventional "dotted methods" but overloaded like top-level functions. Because they are overloaded like top-level functions, the programmer can use the component system to exercise the same fine-grained control over export and import of individual functional method declarations and definitions. (For this reason, and because they address the binary method problem nicely, we often find ourselves preferring functional methods to dotted methods.)

Our purpose here is to explain the Fortress type system, the Meet Rule, the Fortress component system, the `asif` keyword, and Fortress functional methods, and then to illuminate how they interact to solve the problems of method ambiguity, binary methods, super invocation, fine-grained namespace control, and separate compilation.

The specific novel contribution of this work is to present a strategy for implementing symmetric multimethod dispatch in a statically typed language in which the program can be divided into components that can be separately type-checked and separately compiled, *such that the components provide complete namespace control over all top-level names and definitions*; names of not only traits and objects, but also variables and functions and functional methods, and even individual definitions, may be selectively exported, imported, and renamed; the Meet Rule makes possible a distributed implementation of multimethod dispatch that enables separate compilation. The strategy is presented as a multiphase rewriting process; the source language is a stripped-down version of Fortress, and the target language is similar to, but simpler than, the Java programming language and is readily supported by the Java Virtual Machine. The demonstrated rewriting strategy is a practical basis for separate compilation and is easily extended to explicitly type-parameterized methods and functions. We also comment on opportunities for optimizing the resulting target-language code.

2. Source Language Characteristics

The Fortress programming language integrates traditional mathematical notation into an object-oriented framework based on traits with multiple inheritance, overloading (of both methods and functions) resolved by symmetric dynamic dispatch, static types, and separately compiled modules. An innovation of particular interest is *functional methods*, which (like conventional "dotted methods") are declared within traits and may be inherited, but are invoked by ordinary function calls (or mathematical operator syntax) rather than conventional "dotted method calls," and therefore compete in overloading resolution with ordinary function declarations. A component/API system governs visibility of traits, objects, functions, and variables, but when a function definition is invoked, it is desirable to regard overriding functions and methods that are visible to the callee as part of its implementation, even though they may not be visible to the caller. We elaborate on all these points in the following subsections.

2.1 Traits, Objects, Methods, and Functions

Fortress organizes objects into a multiple-inheritance hierarchy based on *traits* [21, 19]. It may be helpful for the reader familiar with the Java language [15] to think of a

Fortress trait as a Java interface that can also contain concrete method definitions, and to think of a Fortress object declaration as a Java final class. Fortress objects inherit only from traits, not other objects, and fields are not inherited. Both traits and objects may contain (concrete) method definitions, and may inherit method definitions from multiple supertraits (traits that they *extend*). Traits may also contain (abstract) method declarations, which impose upon objects that inherit them (directly or indirectly) the requirement to provide matching concrete definitions. Thus all traits and objects form a multiple inheritance hierarchy, a partial order defined by the `extends` relationship, in which all objects are at the leaves. As in the Java language and similar object-oriented languages that have static type systems, the name of a trait or object serves to name a type, which is the set of all objects at or below the named position in the trait hierarchy. In addition to objects (all of which are of type Object, as in the Java language), Fortress has *arrow types* $D \to R$ (the types of functions) and *tuples* written as $(e_1, e_2, \ldots, e_n)$ whose types form a partial order in the usual conjunctive elementwise manner: $(T_1, T_2, \ldots, T_m) \trianglelefteq (U_1, U_2, .., U_n)$ if and only if $m = n$ and, for all $1 \leq k \leq m$, $T_k \trianglelefteq U_k$. Objects, functions, and tuples are all *values* and have type Any.

Like C++ [23] (and unlike the Java language [15] and C# [12]), Fortress provides not only methods associated with objects but also functions not associated with any object; moreover, Fortress function definitions and declarations may be either top-level or local to a block. (We use the term *definition* to refer to a syntactic construct that defines a variable, function, or method, and furthermore contains an executable expression to supply the value of the variable, invoked function, or invoked method. The term *declaration* applies whether or not such an expression is present.)

As in many previous languages, both functions and methods may be *overloaded*; that is, there may be many methods within an object, or many functions declared within the same scope, that have the same name. This raises the issue of *overload resolution*: given a method call or function call, how does the language determine which definition(s) should be invoked? If a single definition is to be chosen and invoked, then we call this determination process the *dispatch mechanism*. A typical dispatch mechanism identifies a pool of candidate definitions that are *accessible* and *applicable*, then selects from that pool the one that is *most specific*. Many dispatch mechanisms have been explored in the literature, and they differ in their definitions of "accessible" and "applicable" and "most specific"; in particular, they may differ in whether static type information, dynamic (run-time) information about the arguments, or both are brought to bear in defining each of these three terms. If static information is used, then the dispatch mechanism may have two stages, one performed at some time prior to program execution (such as compile time) and one performed during program execution. For example, the Java language uses a *single dynamic dispatch* mechanism for method calls, in which both static and dynamic information about the receiver object is used but only static information about the other arguments is used; a method signature is chosen at compile time based on all the static information, and further dispatch occurs at run time based on the actual class of the receiver object (which may turn out to be more specific than the static type of the receiver object expression). As another example, the Common Lisp Object System (CLOS) [13, 4, 22, 14] performs *asymmetric multiple dynamic dispatch* in which only dynamic information is used, but for all arguments, not just one designated "receiver" argument. (While Common Lisp used the term "generic function," nowadays that term is usually used to refer to a function that is polymorphic by virtue of having explicit static type parameters, and the term *multimethod* is used to refer to a set of methods with a dispatch discipline that takes into account dynamic information from more than one argument.) While CLOS performs dynamic dispatch on all arguments, the arguments are treated asymmetrically in that the most specific method is chosen by testing the parameter types of the candidate methods in a specific order (normally left-to-right, though this order can be modified for a particular overload set by an explicit declaration). In contrast, Fortress has the behavior of *symmetric multiple dynamic dispatch*: the method ultimately chosen depends on the ilks of *all* the arguments. (We use the term *ilk* for what is sometimes confusingly called a "run-time type"; just as the Java language has had the slogan "Variables have types, objects have classes" [15, §4.5.5] so Fortress has the slogan "Expressions have types, values have ilks." Among all the types to which a given object might belong, its ilk is (necessarily) the most specific.) However, Fortress also makes use of static type information to perform part of the dispatch at compile time, so it has a two-stage dispatch process. (As we will see, this two-stage process is designed to have the same behavior as a single-stage, fully dynamic dispatch, so the use of static information serves purely as a performance optimization.) Moreover, which methods are accessible depends in part on visibility constraints that may be imposed by dividing a program into components (see Section 2.3). One goal of this paper is to illuminate all these details.

Languages also differ in whether a method or function call can be *ambiguous* in the presence of overloading, and in how ambiguity is detected and handled. Assume that `String` is a subtype of `Object`. In the Java language, for example, one may declare two methods as follows:

```
int gnard(a: String, b: Object) { return 1; }
int gnard(a: Object, b: String) { return 2; }
```

Whatever the type of the receiver object `x`, we may observe that a method call `x.gnard("foo", "quux")` having two strings as arguments will be ambiguous, because both methods are applicable and neither is more specific than the other. As another example, consider three Java interfaces named `A`, `B`, and `C`—where `C` extends both `A` and `B`—and then consider these three methods occurring in some class:

```
void zam(p: C) { System.out.println(jax(p)); }
int jax(q: A) { return 1; }
int jax(q: B) { return 2; }
```

Although the Java language does not have multiple inheritance, it does have multiple supertypes (through the interface hierarchy), and this allows the construction of a method call `jax(p)` that is type-correct but ambiguous. *The Java Language Specification* [15, §15.11.2.2] stipulates that it is permissible to declare such sets of methods, but it is a static error for a program to contain a method call that will be ambiguous in their presence. In contrast, Fortress stipulates that an ambiguous set of overloaded method or function declarations in itself constitutes a static error, even if no call to such a method or function appears in the program.

There is yet another distinction to be made: some languages with multiple inheritance, such as MultiJava [10, 11] and Scala [20], have *asymmetric inheritance*, in which one superclass or supertrait may be treated differently from another based on a criterion such as order of declaration, and this asymmetric treatment may affect the definition of which method is considered "most specific" by the dispatch mechanism (for example, ties might be broken by choosing the method inherited from the supertrait declared earlier). Fortress, in contrast, uses *symmetric inheritance*; methods are inherited on an equal footing from all supertraits. This is another possible source of method ambiguity: two methods with apparently identical signatures may be inherited from two different supertraits. For example, suppose that we have three Fortress traits as follows:

```
trait C extends { A, B }
  zam(): () = println(self.jax())
end

trait A
  jax(): ℤ32 = 1        ❀ ℤ32 is the type of 32-bit integers
end

trait B
  jax(): ℤ32 = 2
end
```

This is similar to the previous example. Because definitions of method jax are inherited symmetrically by trait C, it's a tie: it is not clear which one should be invoked by the method call `self.jax()` in method zam. (By the way, `self.jax()` could have been written more succinctly as simply $jax()$, just as in the Java language; we chose the longer form for this example for the sake of clarity.) So this example is a static error in Fortress and will be rejected by the compiler (indeed, it would be rejected even if the method definition for zam were removed).

Fortress stipulates that such ties must be broken by the programmer. Following Castagna *et al.* [6], Fortress requires that the signatures of all the functions or methods in an overload set form a *meet-bounded lattice*—we call this the Meet Rule. Simply put, for every pair of accessible signatures for which it might be possible to construct an ambiguous call (that is, both are applicable and neither is more specific than the other), there must be a third accessible signature that (a) is more specific than each of the other two signatures, and (b) is also applicable to the call, and therefore will be chosen unambiguously over the other two. We can repair the previous example by introducing an appropriate definition of jax into trait C. This might simply supply a value on its own:

$$jax(): \mathbb{Z}32 = 47$$

or choose to direct control to one of the inherited definitions:

$$jax(): \mathbb{Z}32 = (\texttt{self asif } \mathrm{B}).jax()$$

or might even use more than one of the inherited definitions:

$$jax(): \mathbb{Z}32 = (\texttt{self asif } \mathrm{A}).jax() + (\texttt{self asif } \mathrm{B}).jax()$$

(As we shall see, the Fortress `asif` construction addresses the same issues that `super` does in the Java language.)

In addition to the Meet Rule, Fortress imposes the usual Result Subtype Rule: for any pair of accessible functions or methods in an overload set, if the first is more specific than the second, then the result type of the first must be a subtype of the result type of the second. This rule is necessary to maintain type soundness.

Fortress also has parametric polymorphism of traits, objects, methods, and functions; any of these may have static type parameters, and references to them may supply explicit static arguments, which may be types, boolean values, integers, or operator symbols. However, we do not address parametric polymorphism of traits and objects in this paper, and we discuss parametric polymorphism of functions and methods only briefly, in Section 8.

2.2 Functional Methods

As the Fortress design team has previously reported [3], Fortress has not only *dotted methods* that are declared much as in the Java language and are invoked with a *dotted method call* of the form:

$$receiver.methodName(e_1, e_2, \ldots, e_n)$$

but also *functional methods*. The declaration of a functional method differs from that of a dotted method in that exactly one of the parameter $name: \mathrm{Type}$ declarations in the header is replaced with the keyword `self`; in effect, that argument position, rather than a separate expression "before the dot," is treated as the receiver. This feature easily solves the "binary method problem" [5]: Suppose that we define a trait Point having real-valued fields x and y and a method that tests equality of points by comparing their x and y components:

```
trait Point
  x: ℝ
  y: ℝ
  equal(self, other: Point): Boolean =
    (x = other.x) ∧ (y = other.y)
end
```

(In Fortress, traits cannot really have fields, but the field declaration syntax is construed as implicitly declaring abstract getter methods:

```
getter x(): ℝ
getter y(): ℝ
```

and a field access expression such as $p.x$ is construed to mean $p.x()$ if x names a getter method. Field declarations in objects, as opposed to traits, *do* declare actual fields, and furthermore implicitly declare concrete getter methods that will fetch the values of those fields.)

Now we wish to extend this trait to create a new trait ColorPoint that has an additional field c of type Color, and we ask that equality comparison of colored points should also test whether they have the same color. We may write:

```
trait ColorPoint extends Point
  c: Color
  equal(self, other: ColorPoint): Boolean =
    (x = other.x) ∧ (y = other.y) ∧ (c = other.c)
end
```

There are now two points to observe. First: ColorPoint is not only a subtrait of Point but in fact a subtype of Point. In Fortress, subtraits are always subtypes because of various constraints in the language specification as to what programs are valid, but we say this explicitly because in the example language of Bruce et al. [5] subclasses are not necessarily subtypes—indeed, they were exploring the very question of what constraints are necessary for a subclass to be a subtype. In this example, unlike that in section 2.1 of Bruce et al., ColorPoint not only provides an explicit definition of method *equal* but also inherits a definition of method *equal* from Point, just as it inherits the abstract getter methods for x and y (and therefore any object that extends ColorPoint inherits an obligation to implement those getter methods, as well as inheriting the obligation to implement a getter method for c). As a result, the functional method *equal* may correctly be called with any mixture of Point and ColorPoint arguments; if both arguments are colored points, then the definition in ColorPoint will be used, and otherwise the definition in Point will be used. Therefore Fortress solves the classic ColorPoint "binary method problem" as presented in Bruce et al.

Second: Using the definitions just shown, $equal(p, cp)$ simply ignores the color of cp when comparing point p to colored point cp. We might want to control this behavior more finely by providing explicit method definitions for the cases where a point is compared with a colored point—for example, to treat an ordinary point as if its color were Purple. We might try using functional methods this way, without having to modify the definition of Point:

```
trait ColorPoint extends Point      ❀ incorrect example
  c: Color
  equal(self, other: ColorPoint): Boolean =
    (x = other.x) ∧ (y = other.y) ∧ (c = other.c)
  equal(self, other: Point): Boolean =
    (x = other.x) ∧ (y = other.y) ∧ (c = Purple)
  equal(other: Point, self): Boolean = equal(self, other)
end
```

The third definition of *equal* in ColorPoint has the `self` keyword in the second parameter position, and therefore is considered applicable to invocations that have an argument of type ColoredPoint in the second argument position (and an argument of type Point in the first position)—so far, so good. The problem is that the Meet Rule in Fortress must statically forbid overloadings in which there are two functional methods, neither more specific than the other and neither excluding the other, in which the `self` parameter appears in different argument positions. (The problem is that the compiler needs to decide statically which argument position to treat as the receiver, without knowing the precise set of dynamically available overloadings.) As a result, the current design of Fortress does *not* solve this particular extended version of the binary method problem, and we regard this as an area for future research.

We furthermore observe that other traits, completely unrelated to points and colored points, might also define functional methods named *equal*, and there might also be top-level definitions of functions named *equal*. All these constitute one large overload set, and as long as the Meet Rule and the Result Subtype Rule (governing consistency of return types when one definition is more specific than another) are satisfied, overloading ambiguity will not occur.

To guarantee this lack of ambiguity, sometimes it is necessary to declare that two traits *exclude* each other—that is, no object can belong to both traits). To see this, consider the example that originally motivated the introduction of functional methods into Fortress. We have a Vector trait that implements typical mathematical vector operations such as scalar multiplication, dot product, and cross product:

```
trait Vector
  multiply(v: ℝ, self): Vector = ...
  dotProduct(self, v: Vector): ℝ = ...
  crossProduct(v: Vector, self): Vector = ...
end
```

Now we wish to define a Matrix trait that implements not only scalar-matrix multiplication and matrix-matrix multiplication but also matrix-vector multiplication and vector-matrix multiplication, all without modifying the existing Vector trait. Moreover, unlike the example of Point and ColorPoint, it is not appropriate for Matrix to be a subtrait of Vector. If we simply write:

```
trait Matrix
  multiply(v: ℝ, self): Matrix = ...
  multiply(self, v: Matrix): Matrix = ...
  multiply(self, v: Vector): Vector = ...
  multiply(v: Vector, self): Vector = ...
end
```

the Meet Rule is not satisfied, because it is possible that some object vm might extend both Vector and Matrix, and then the invocation $multiply(4, vm)$ would be ambiguous because two definitions of scalar multiplication would apply, neither more specific than the other. A solution is to declare that Matrix excludes Vector, so that no such object can exist:

```
trait Matrix excludes { Vector }
  multiply(v: ℝ, self): Matrix = ...
  multiply(self, v: Matrix): Matrix = ...
  multiply(self, v: Vector): Vector = ...
  multiply(v: Vector, self): Vector = ...
end
```

Thus, while Fortress cannot currently solve the extended ColorPoint example (because ColorPoint inherits from Point and therefore does not exclude it), it handles the Vector/Matrix version of the binary method problem quite nicely (because Matrix excludes Vector). This suggests a partial solution to the extended ColorPoint problem: if we know about a particular other trait, say InvisiblePoint, that extends Point, and is known not to have any instances in common with ColorPoint, then we can write *equal* methods in ColorPoint to handle all comparisons with InvisiblePoint without having to modify InvisiblePoint:

```
trait ColorPoint extends Point excludes { InvisiblePoint }
  c: Color
  equal(self, other: ColorPoint): Boolean =
    (x = other.x) ∧ (y = other.y) ∧ (c = other.c)
  equal(self, other: InvisiblePoint): Boolean =
    (x = other.x) ∧ (y = other.y) ∧ (c = Purple)
  equal(other: InvisiblePoint, self): Boolean =
    equal(self, other)
end
```

This overloading of *equal* satisfies the Meet Rule because the signature (ColorPoint, InvisiblePoint) and the signature (InvisiblePoint, ColorPoint) exclude each other.

In our early report [3], Fortress allowed functions to be overloaded with other functions and functional methods to be overloaded with other functional methods. Since then Fortress has been changed to allow functions and functional methods to participate in the same overload set. One goal of this paper is to explain exactly how that works. A necessary restriction is that there must not be any top-level function whose signature is more specific than that of a functional method within the same overload set.

2.3 Components and APIs

Just as the Java language allows a program to be divided into *compilation units* that can be separately compiled, and also has *packages* that provide a modicum of namespace control, Fortress allows a program to be divided into *components*. Interfaces between components are described by *APIs*. Code in one component cannot refer directly to code in another component; rather, any component or API can *import* identifiers from one or more APIs, and any component can *export* one or more APIs. Each API must be exported by exactly one component.

Fortress also has mechanisms for bundling a set of components into a single larger component, and this process may be carried out recursively, thus providing namespace control for the names of components and APIs. The details of this are beyond the scope of this paper; here we will simply assume that the names of all components are distinct and the names of all APIs are distinct (but the name of a component may be the same as the name of an API).

One goal of this paper is to explain how overload sets and overload resolution interact with the namespace control imposed by components and APIs in Fortress.

A Fortress API can contain declarations of top-level variables, top-level functions, traits, and objects; trait and object definitions may contain (abstract) declarations of fields, dotted methods, and functional methods. Here is a simple example of an API:

```
api Geometry

trait Point
  x: ℝ
  y: ℝ
  equal(self, other: Point): Boolean
end

makePoint(x: ℝ, y: ℝ): Point

end Geometry
```

An API may be implemented by one or more components, but only one implementation may be used in any given complete program. Here is a possible implementation of the Geometry API:

```
component QuuxGeometry
export Geometry

trait Point
  x: ℝ
  y: ℝ
  equal(self, other: Point): Boolean =
    (x = other.x) ∧ (y = other.y)
end

makePoint(x: ℝ, y: ℝ) = CartesianPoint(x, y)

object CartesianPoint(x: ℝ, y: ℝ) extends Point end

object PolarPoint(ρ: ℝ, θ: ℝ) extends Point
  getter x() = ρ cos θ
  getter y() = ρ sin θ
end

end QuuxGeometry
```

Note that ρ and θ are legitimate identifiers in Fortress, as they are (in principle) in the Java language. The object PolarPoint is not exported through the Geometry, but could be exported as part of another API or used as part of the internal workings of the component (not shown).

We might then have another component that can be run as a "main program" (because it exports the API Executable, which declares a top-level function named *run*), which also imports Color and the names of three colors from ColorPackage (not shown) and imports "whatever is needed" (the ellipsis ... in the `import` statement is actually part of Fortress "import on demand" syntax) from the Geometry API:

```
component MyProgram
import ColorPackage.{ Color, Orange, Green, Purple }
import Geometry.{ ... }
export Executable

trait ColorPoint extends Point
  c: Color
  equal(self, other: ColorPoint): Boolean =
    (x = other.x) ∧ (y = other.y) ∧ (c = other.c)
  equal(self, other: Point): Boolean =
    (x = other.x) ∧ (y = other.y) ∧ (c = Purple)
  equal(other: Point, self): Boolean = equal(self, other)
end

object ColorPointObject(x: ℝ, y: ℝ, c: Color)
      extends Point
end

makeColorPoint(x: ℝ, y: ℝ, c: Color) =
  ColorPointObject(x, y, c)

run(): () = do
  p = makePoint(4.7, 5.2)
  cp = makeColorPoint(4.7, 5.2, Purple)
  println(equal(p, cp))
end

end MyProgram
```

The API can be separately compiled, after which the two components can be separately compiled; then a linking process binds the two components together for execution, verifying that every API used by the program has an implementing component.

Now that we have briefly explained and illustrated the use of components and APIs in Fortress, we need to bring out an important interaction between the component (namespace control) mechanism and overloading.

Let the names $\mathbb{Z}$ and $\mathbb{N}$ name the types "integers" and "natural numbers" (as they normally do in Fortress), where $\mathbb{N}$ is a subtype of $\mathbb{Z}$. Then consider this API:

```
api IntegerToString

trait ℤ
  getter asString(): String
end

end IntegerToString
```

It might be implemented by this component (shown only in part; it might well define other methods and export other, more complex APIs in addition to IntegerToString):

```
component IntegerToString
export IntegerToString

trait ℤ
  getter asString() = "–" ‖ (− self).asString
end

trait ℕ extends ℤ
  getter asString() =
    if  self < 10 then "0123456789"[self : self + 1]
    else
      q = self ÷ 10
      r = self − 10 q
      q.asString ‖ r.asString
    end
end

end IntegerToString
```

Without going into implementation details, let us simply note the members of $\mathbb{Z}$ are the members of $\mathbb{N}$ plus all the negative integers. The getter *asString* has been implemented as an overloaded method: if the receiver object is nonnegative, then the definition in trait $\mathbb{N}$ will be used because it is more specific, but if the receiver object is negative, then the definition in trait $\mathbb{Z}$ will be used because the one in $\mathbb{N}$ is not applicable. Both definitions are part of the intended algorithm for converting a value of type $\mathbb{Z}$ to a string.

But the API mentions only trait $\mathbb{Z}$ and its one declaration of *asString*, so only one declaration of *asString* is exported from component IntegerToString. If that exported declaration is to be meaningful, it must somehow carry with it the implementation functionality of *both* definitions of *asString*. And in fact it does, thanks to the customary behavior of "dotted method" inheritance and the rules of overloading (which in other languages would be called overriding, but in Fortress overriding is merely a special case of overloading because the type of the receiver object is taken into account in comparing method signatures). So this example should be completely unsurprising so far.

But now let us consider the same example using top-level functions rather than dotted methods. The API is:

```
api IntegerToStringFunction

asString(x: ℤ): String

end IntegerToStringFunction
```

and the component is:

```
component IntegerToStringFunction
export IntegerToStringFunction

asString(x: ℤ) = "–" ‖ (−x).asString

asString(x: ℕ) =
  if x < 10 then "0123456789"[x : x + 1] else
    q = x ÷ 10
    r = x − 10 q
    q.asString ‖ r.asString
  end

end IntegerToStringFunction
```

When another component imports the API, it imports only a declaration of *asString* that accepts arguments of type $\mathbb{Z}$, but clearly the functionality exported should somehow include both definitions. So it won't do to say glibly "only the first definition is exported"; a deeper explanation is required. It is this: every function definition and every method defini-

tion is, in effect, required to defer (that is, possibly dispatch) to other functions or methods that are accessible within that component, applicable to the arguments received, and more specific than the function or method actually called.

To see why the words "within that component" are needed in the previous statement, consider this importing component:

```
component TrickyUser
import IntegerToStringFunction.{ asString }
export Executable

asString(x: ℕ) = "+" ‖ asString(x asif ℤ)

run() = println (153).asString

end TrickyUser
```

This user expects the output to be "+153"; from the point of view of this component, there are two accessible definitions of *asString*, one defined within the component that accepts natural numbers and one imported from the IntegerToStringFunction API that accepts any integer. The first definition handles natural numbers by asking the imported definition to convert the number to a string and then prepending a plus sign (the `asif` keyword indicates that the function invocation *asString*(*x* `asif` ℤ) should treat the argument as having type ℤ rather than ℕ for dispatch purposes). That is all well and good; the real point is that it is very important that this local definition of *asString* in component TrickyUser mustn't somehow override the second definition of *asString* in component IntegerToStringFunction, because that would completely destroy the modularity that components are intended to provide. Therefore a model that says, "Component IntegerToStringFunction exports a generic function named *asString* that consists of a set of definitions (an overload set) for *asString*, and component TrickyUser then further overloads this generic function to produce a new overload set for *asString* for use within component TrickyUser" will not work. It does not suffice to simply merge overload sets; we want a model that will preserve the structure of per-component accessibility. We have already mentioned the model that works for Fortress: every function definition and every method definition must defer to other functions or methods that are accessible within that component, applicable to the arguments received, and more specific than the function or method actually called.

Let us revisit the example one more more time, but using functional methods rather than dotted methods or top-level functions. The API is:

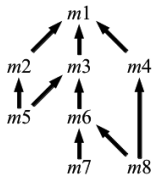
```
api IntegerToStringFunctionalMethod

trait ℤ
  asString(self): String
end

end IntegerToStringFunctionalMethod
```

and the component is:

```
component IntegerToStringFunctionalMethod
export IntegerToStringFunction
export IntegerToStringFunctionalMethod

trait ℤ
  asString(self) = "-" ‖ (− self).asString
end

trait ℕ extends ℤ
  asString(self) =
    if  self < 10 then "0123456789"[self : self + 1]
    else
      q = self ÷ 10
      r = self − 10 q
      q.asString ‖ r.asString
    end
end

end IntegerToStringFunctionalMethod
```

Note that component IntegerToStringFunctionalMethod exports both the IntegerToStringFunctionalMethod API and the IntegerToStringFunction API. Both are correct. The IntegerToStringFunctionalMethod API in effect exports a top-level function named *asString* and also the trait ℤ; the IntegerToStringFunctionalMethod API exports only a top-level function named *asString*. The component effectively implements both. Functional methods are inherited like dotted methods—most importantly, *abstract* functional methods, carrying the obligation to provide concrete implementations, are inherited like abstract dotted methods. But they are overloaded in the same per-component global namespace as top-level functions, and the dispatch model that works for top-level functions while preserving component modularity also works for functional methods.

2.4 The Deferral Lattice



Consider a set of eight overloaded method (or function) definitions, which for convenience we will call *m1* through *m8* even though they all actually define the same name *m*. Now suppose that *m2*, *m3*, and *m4* are each more specific than *m1*; *m5* is more specific than *m2* and *m3*; *m6* more specific than *m3*; *m7* more specific than *m6*; and *m8* more specific than *m6* and *m4* (see figure). The complete "more specific" relation is of course the transitive closure of the stated relationships, so that, for example,*m8* is also more specific than *m1* and *m3*.

The idea behind our implementation strategy is that if, for a given invocation of *m*, the compiler determines that *m1* is the statically most specific definition, then at run time *m1* must defer to more specific definitions. The key insight is that, while it is *correct* for m1 to consider deferring to each of *m2* through *m8*, it is *sufficient* for *m1* to defer only to *m2*, *m3*, and *m4*, because each of them has the same obligation to defer. Moreover, if *m1* has more than one choice, the choice does not matter, thanks to the Meet Rule. For example,

suppose that at run time *m8* is actually the dynamically most specific applicable definition; then *m2* must not be applicable, and *m3* and *m4* must be applicable. It does not matter whether *m1* hands off responsibility to *m3* or *m4*; if the eventual correct choice is *m8*, then repeated deferral will eventually transfer control down the lattice to *m8*.

Therefore our strategy is to rewrite each function or method definition into two: the extra function is considered the primary entry point and implements the lattice deferral strategy. For example, *m1* is augmented with a primary entry point *m1entry* that has roughly this structure:

m1entry: if *m2* is applicable then call *m2entry*
else if *m3* is applicable then call *m3entry*
else if *m4* is applicable then call *m4entry*
else call *m1*

Similarly, *m3* is augmented with *m3entry*:

m3entry: if *m5* is applicable then call *m5entry*
else if *m6* is applicable then call *m6entry*
else call *m3*

For the sake of uniformity, *m7* is augmented with *m7entry*:

m7entry: call *m7*

Routine inlining optimizes away such trivial entry points.

As we shall see in Section 5, in practice we rewrite each definition into more than two definitions, in order to handle the `asif` construction and some of the tricky points of deferring dotted methods, but the basic idea of deferral down the lattice of the "more specific" relation applies to both dotted methods and top-level functions; the details differ because of the differing visibility rules for methods and functions. On the other hand, functional methods are handled by first rewriting them into a collection of top-level functions and dotted methods.

The notion of being "more specific" comes into play only after accessibility has been considered. Accessibility is in turn governed by the component system. As a result, while there is a single dispatch lattice for all dotted methods named m, there is a separate dispatch lattice for top-level functions named f within each component in which some function named f is visible. Importing some definitions of function f from component A to component B causes corresponding entry points in component A to become part of the dispatch lattice for f in component B.

3. The Source Language

Here we summarize the source language, which is a subset of Fortress (in particular, variables declared at the top level of a component or API are omitted as not salient to our focus here, and declarations and uses of operator symbols are assumed to have already been converted to declarations of ordinary functions or functional methods and to ordinary function calls). Please refer to Figure 1.

A program in the source language is a collection of components and APIs; their order does not matter.

Each API contains declarations of traits and functions; their order does not matter. (Actual Fortress APIs also allow declarations of objects and variables, but the bookkeeping related to those is not relevant to the theme of this paper.) An API may also import traits and functions from other APIs, thus allowing aggregation of APIs.

Each component contains definitions of traits, objects, top-level functions, and top-level variables, as well as (abstract) declarations of top-level functions. A component may also import names of traits and functions from APIs; their implementations will be supplied by other components. A component may export one or more APIs; in order to export an API, a component must provide a (concrete) definition corresponding to every declaration appearing in the API. (A Fortress component can "provide" a definition either by containing an actual definition or by importing it from another API; for present purposes we will require that the component itself contain the definition.) Importing a trait from an API enables invocation of dotted methods declared within that trait in the API.

Import declarations can rename traits and functions as they are imported, so that the referring component can use a name different from the one specified in the API.

A trait declaration may contain declarations and (when within a component) definitions of dotted methods and functional methods. A trait declaration may also contain what looks like a declaration of a field, but (as we shall see) this is really regarded as a requirement that appropriate getter and setter methods be defined by any implementing object. A trait may also include any or all of three clauses that relate it to other traits and objects: (1) A trait A may *extend* one or more other traits $B_1, B_2, \ldots, B_p$, in which case for each $1 \leq j \leq p$, A is a subtype of B_j, and A inherits all the methods of B_j that it does not override. (2) A trait A may *comprise* one or more other traits $C_1, C_2, \ldots, C_q$, in which case it is a static error for any trait or object D to extend A unless D is a subtype of (and possibly equal to) at least one C_j for some j such that $1 \leq j \leq q$; furthermore, each C_j must be declared in the same component as A and must extend A. (3) A trait A may *exclude* one or more other traits $E_1, E_2, \ldots, E_r$, in which case it is a static error if any trait or object F extends A and also extends E_j for any j such that $1 \leq j \leq r$. (The `comprises` and `excludes` clauses do not figure prominently in this paper, but we note that they are very important in practice in a multiple-inheritance framework in order to indicate that certain edges *cannot* occur in the type hierarchy. Without these clauses, it would be difficult or impossible to get certain useful method overloadings to conform to static error checks.)

Object declarations may contain definitions of dotted methods, functional methods, and variables; variable definitions within an object declaration are regarded as fields of the object. An object declaration may contain an `extends` clause that mentions supertraits, just as for a trait declaration; in effect, an `object` declaration declares a trait of the

Program ::= (*API* | *Component*)*
API ::= `api` *ApiName Import** (*ApiTraitDecl* | *FunctionDecl*)* `end` *ApiName*
Component ::= `component` *ComponentName Import** *Export*+ *ComponentItem** `end` *ComponentName*
ComponentItem ::= *TraitDefn* | *ObjectDefn* | *FunctionDecl* | *FunctionDefn* | *VariableDefn*
Import ::= `import` *ApiName* `.` `{` *ImportItemList* `}`
ImportItemList ::= *ImportItem* (`,` *ImportItem*)*
ImportItem ::= *TraitName* (↦ *TraitName*)? | *FunctionName* (↦ *FunctionName*)?
Export ::= `export` *ApiName*
ApiTraitDecl ::= `trait` *TraitName Extends*? *Comprises*? *Excludes*? *ApiTraitItem** `end`
ApiTraitItem ::= *VariableDecl* | *DottedMethodDecl* | *FunctionalMethodDecl*

TraitDefn ::= `trait` *TraitName Extends*? *Comprises*? *Excludes*? *TraitItem** `end`
ObjectDefn ::= `object` *ObjectName* (`(` *ParameterList*? `)`)? *Extends*? *ObjectItem** `end`
Extends ::= `extends` `{` *TraitNameList* `}`
TraitNameList ::= *TraitName* (`,` *TraitName*)*
Comprises ::= `comprises` `{` *TypeNameList* `}`
Excludes ::= `excludes` `{` *TypeNameList* `}`
TraitItem ::= *ApiTraitItem* | *DottedMethodDefn* | *FunctionalMethodDefn*
ObjectItem ::= *VariableDefn* | *DottedMethodDefn* | *FunctionalMethodDefn*

DottedMethodDecl ::= (`getter` | `setter`)? *MethodName* `(` *ParameterList*? `)` `:` *Type*
DottedMethodDefn ::= *DottedMethodDecl* = *Expression*
FunctionalMethodDecl ::= *FunctionName* `(` *SelfParameterList* `)` `:` *Type*
FunctionalMethodDefn ::= *FunctionalMethodDecl* = *Expression*
FunctionDecl ::= *FunctionName* `(` *ParameterList*? `)` `:` *Type*
FunctionDefn ::= *FunctionDecl* = *Expression*

ParameterList ::= *VariableDecl* (`,` *VariableDecl*)*
SelfParameterList ::= (*VariableDecl* `,`)* `self` (`,` *VariableDecl*)*
VariableDecl ::= `var`? *VariableName* `:` *Type*
VariableDefn ::= *VariableDecl* (`=` | `:=`) *Expression* | *SimpleBinding* = *Expression*
SimpleBinding ::= *VariableName* | `(` *SimpleBindingList*? `)`
SimpleBindingList ::= *SimpleBinding* (`,` *SimpleBinding*)*

Type ::= *TypeName* | `(` *TypeList*? `)` | *Type* → *Type*
TypeList ::= *Type* (`,` *Type*)*
TypeName ::= *TraitName* | *ObjectName*
TypeNameList ::= *TypeName* (`,` *TypeName*)*

Expression ::= *VariableName* | *Literal* | `self` | `(` *ExpressionList*? `)` | *Assignment* |
FunctionCall | *DottedMethodCall* | *FieldAccess* | *FieldAssignment* | *IfThenElseEnd* | *DoEnd*
ExpressionList ::= *Expression* (`,` *Expression*)*
Assignment ::= *VariableName* := *Expression*
FunctionCall ::= *FunctionName* `(` *CallExprList*? `)`
DottedMethodCall ::= *CallExpr* `.` *MethodName* `(` *CallExprList*? `)`
CallExpr ::= *Expression* (`asif` *Type*)?
CallExprList ::= *CallExpr* (`,` *CallExpr*)*
FieldAccess ::= *CallExpr* `.` *VariableName*
FieldAssignment ::= *CallExpr* `.` *VariableName* := *CallExpr*
IfThenElseEnd ::= `if` *Expression* `then` *StatementList* (`elif` *Expression* `then` *StatementList*)* (`else` *StatementList*)? `end`
DoEnd ::= `do` *StatementList* `end`
StatementList ::= (*FunctionDefn* | *VariableDefn* | *Expression*)* *Expression*

Figure 1. Grammar for source language

Program ::= (*FunctionDefn* | *InterfaceDefn* | *ClassDefn* | *VariableDefn*)*
InterfaceDefn ::= `interface` *InterfaceName Extends*? *DottedMethodDecl** `end`
ClassDefn ::= `class` *ClassName Extends*? *Constructor ClassItem** `end`
Extends ::= `extends` { *InterfaceNameList* }
InterfaceNameList ::= *InterfaceName* (`,` *InterfaceName*)*
Constructor ::= `constructor` *ClassName* `(` *ParameterList*? `)` = *Expression*
ClassItem ::= *VariableDefn* | *DottedMethodDefn*

DottedMethodDecl ::= *MethodName* `(` *ParameterList*? `)` : *Type*
DottedMethodDefn ::= *DottedMethodDecl* = *Expression*
FunctionDefn ::= *FunctionName* `(` *ParameterList*? `)` : *Type* = *Expression*

ParameterList ::= *VariableDecl* (`,` *VariableDecl*)*
VariableDecl ::= `var`? *VariableName* : *Type*
VariableDefn ::= *VariableDecl* (= | :=) *Expression* | *SimpleBinding* = *Expression*
SimpleBinding ::= *VariableName* | `(` *SimpleBindingList*? `)`
SimpleBindingList ::= *SimpleBinding* (`,` *SimpleBinding*)*

Type ::= *TypeName* | `(` *TypeList*? `)` | *Type* → *Type*
TypeList ::= *Type* (`,` *Type*)*
TypeName ::= *InterfaceName* | *ClassName*

Expression ::= *VariableName* | *Literal* | `self` | `(` *ExpressionList*? `)` | *Assignment* |
TypeLiteral | *TypeTest* | *Construction* | *Conjunction* |
FunctionCall | *DottedMethodCall* | *FieldAccess* | *FieldAssignment* | *IfThenElseEnd* | *DoEnd*
ExpressionList ::= *Expression* (`,` *Expression*)*
TypeLiteral ::= *Type* `.` `type`
TypeTest ::= *Expression* `instanceof` *Type*
Construction ::= `new` *ClassName* `(` *ExpressionList*? `)`
Assignment ::= *VariableName* := *Expression*
Conjunction ::= *Expression* `&&` *Expression*
FunctionCall ::= *FunctionName* `(` *ExpressionList*? `)`
DottedMethodCall ::= *Expression* `.` *MethodName* `(` *ExpressionList*? `)`
FieldAccess ::= `self` `.` *VariableName*
FieldAssignment ::= `self` `.` *VariableName* := *Expression*
IfThenElseEnd ::= `if` *Expression* `then` *StatementList* (`elif` *Expression* `then` *StatementList*)* (`else` *StatementList*)? `end`
DoEnd ::= `do` *StatementList* `end`
StatementList ::= (*FunctionDefn* | *VariableDefn* | *Expression*)* *Expression*

Figure 2. Grammar for target language

same name that serves as a type name, but there is no need for such a trait to have a `comprises` clause or `excludes` clause because it cannot have subtraits. Each object type implicitly excludes every other object type, and indeed every type that is not its ancestor in the trait hierarchy.

If an object declaration has a parameter list, then there is an implicit top-level constructor function with the same name, and each invocation of the constructor function returns a fresh object whose ilk is the implicit trait of that same name; moreover, each parameter is also a field of the freshly constructed object. This constructor function can participate in an overload set with ordinary functions and functional methods of the same name. If an object declaration has no parameter list, then it represents a *singleton object* whose ilk is the implicit trait with the same name, and there is an implicit top-level variable with the same name whose value is that singleton object.

A dotted method declaration or definition is much as in the Java language, with slightly different syntax. A dotted method with the `getter` keyword is invoked by a field access, and a dotted method with the `setter` keyword is invoked by a field assignment. Such a getter or setter method must not be present in the same trait or object declaration as a field declaration or definition with the same name. Functional method declarations are distinguished syntactically from dotted method declarations by the presence of the keyword `self` in (exactly one place in) the parameter list.

Variable declarations specify a name and a type; if the keyword `var` is present, the variable is mutable. Variable definitions additionally specify an expression that provides

an initial value. The symbol $=$ indicates definition of an immutable variable; the symbol $:=$ (which is also used for assignment) indicates definition of a mutable variable (in which case `var` may optionally appear). As a convenience, within a statement list, a tuple of simple variable names may be bound to the respective elements of a tuple value.

A type may be the declared name of a trait or object, a parenthesized type, a tuple of types, or an arrow type. (The interesting issue of what is the precise type of an overloaded function name when used as a value is beyond the scope of this paper.)

Names of types are in a different namespace from the names of variables and functions; one may have a type and a function, or a type and a variable, with the same name. Definitions and declarations of local function and variables are not permitted to shadow outer definitions or declarations with the same name.

The Fortress parser rewrites operator syntax into function calls in a conventional manner, so there is no need to model operators separately in our source language here.

A function call may invoke either an ordinary function, a functional method, or an object constructor—but if the function call occurs within a trait or object declaration that has or inherits a definition or declaration of a dotted method with the same name, then it is considered to invoke that dotted method with `self` as the receiver. (This "convenience abbreviation" of a dotted method call—allowing the elision of a leading "`self.`" or "`this.`"—will be familiar to programmers who use the Java language [15, §15.11].)

Within an invocation of a dotted method or functional method, an immediate subexpression (that is, an argument expression or receiver expression) may have the form x `asif` T where x is an expression and T is a type; this is used to get the effect of the Java `super` construct, but specifies that T is the specific supertype of interest. If any subexpression of an invocation is an `asif` expression, then the subexpression before the dot (in a dotted method call) or corresponding to the `self` parameter of the statically most specific applicable declaration or definition, which must be a declaration or definition of a functional method (in a function call), must also be an `asif` expression.

Within a block (which for present purposes we may regard as identical to a *StatementList*, though the actual Fortress grammar draws a subtle distinction for the purpose of defining mutually recursive functions), functions and variables may be defined locally.

Identifiers may include any Unicode letter, digit or connecting punctuation, but not dollar signs "$'.

4. The Target Language

Here we summarize the features of the target language that differ from those of the source language. Please refer to Figure 2.

A target-language program consists of a set of definitions of interfaces, classes, top-level functions, and top-level variables. There are no components or APIs. A target-language program is much like a single Java compilation unit.

Interface declarations contain only dotted method declarations; there is no executable code in an interface. There are no functional methods. Methods cannot be overloaded, and `comprises` and `excludes` clauses are unnecessary.

Class declarations are similar to those in the Java language. Every class declaration has a `constructor` declaration; a constructor is invoked by a construction expression beginning with the keyword `new`. Class declarations may contain dotted method definitions, as well as declarations and definitions of fields.

For convenience, the target language includes the short-circuit (that is, conditional) Boolean AND operator `&&`, as well as the `instanceof` operator that can test whether the value of an expression belongs to a specified type. If a variable passes an `instanceof` check in the predicate part of an `if` statement, then the variable is regarded as having that type in the corresponding `then` branch of the `if` statement (thus no cast is required).

An expression of the form $T.\texttt{type}$ is a *type literal* (analogous to an expression of the form $T.\texttt{class}$ in the Java language); its value (which is of type Type) is a run-time reification of the type T. Such an object provides a dotted method $t.isSubtype(u)$ where u is also such an object, which returns $true$ if the type represented by t is a subtype of (possibly the same as) the type represented by u, and otherwise returns $false$. (Compare the method `isAssignableFrom` of the Java class `class`, which provides the converse "is a supertype" test.)

Every value x provides the dotted method $x.ilk()$, which returns a reified type object that represents the ilk of the object, function, or tuple. The reified ilk of a function is of type $\mathrm{ArrowType}$, and provides additional methods such as $argumentType$ and $resultType$, which for a function type $D \to R$ return reified type objects for D and R, respectively. (Reified tuple types also have extra methods, but we will not need them for our exposition here.)

Field accesses and assignments are supported, but only within class declarations, and the expression before the dot must be `self`.

It is convenient to assume that target-language identifiers, in addition to Unicode letters, digits, and connecting punctuation, may also include a character "$' that cannot appear in source-language identifiers, in order to allow construction of "mangled" names that cannot conflict with identifiers originating in source-language code. In what follows, it is assumed that any target-language identifier containing one or more "$' characters and also one or more variables mentioned in the text actually represents an identifier constructed by substituting some string value (a name or the decimal representation of an integer value) for each variable. For example, if we say that $j = 3$ and w stands for the name foo, then $w\$j\$dispatch$ stands for the identifier $foo\$3\$dispatch$.

5. Details of the Translation

The purpose of this section is to explain some of the detailed semantics of the source language outlined in Section 3 by showing how to translate it into the much simpler target language outlined in Section 4.

To summarize, the source language has these features of interest:

a static type system;
traits and objects can extend multiple traits;
multiple inheritance of both dotted and functional methods;
both objects and traits can contain both declarations and definitions of both dotted and functional methods;
fields are declared only in objects;
objects cannot extend other objects;
dotted methods, functional methods, top-level functions, and local functions all support symmetric dynamic multi-method dispatch among overloaded definitions;
an "`asif`" feature to get "`super`"-like behavior in a multiple-inheritance, symmetric-dispatch environment;
a rule that overloaded definitions must form a meet-bounded lattice, thereby making it impossible for any function or method call to be ambiguous;
a system of modular components and APIs that allows controlled import and export of names of individual traits, objects, top-level functions, and top-level variables; and
separate compilation, and therefore separate static type-checking, of components.

and the target language has these features and limitations:

a static type system;
interfaces and classes can extend multiple interfaces;
multiple inheritance of dotted methods
interfaces can contain only abstract dotted method declarations;
classes can contain concrete dotted method declarations;
fields are declared only in classes;
classes do not extend other classes;
no overloading or overriding, and no inheritance of concrete method definitions;
no "`super`" feature of any kind;
field access occurs only within the dotted methods of the declaring class;
dotted methods need dispatch only on the receiver;
top-level functions and local functions are not overloaded and therefore require no dispatch;
one global namespace and no separate compilation;
the minimal reflective ability to obtain a "type object" representing either an arbitrary type or the ilk (run-time type) of an object, and to inquire of two such type objects whether one represents a subtype of the other.

The rewriting does not group a set of overloaded methods or functions into a monolithic "generic" method or function; rather, the implementation of the dispatch process is distributed in a manner that allows selective export and selective optimization. The source language captures essential features of the multimethod dispatch mechanism of the Fortress programming language. The target language is considerably simpler than the Java programming language and is readily supported by the Java Virtual Machine. The demonstrated rewriting is a practical basis for separate compilation and is easily extended to the case of explicitly type-polymorphic methods and functions.

The big picture: trait declarations are rewritten into interface declarations, object definitions are rewritten into class definitions, fields are largely eliminated by using getter and setter methods, and functional method definitions are eliminated by translating them into top-level function definitions that serve as trampolines to corresponding dotted method definitions. Namespaces are eliminated by mangling names to include the name of the defining component.

The source program is first subjected to static analysis, including verification of syntactic well-formedness, static type checking, and verification that every overload set obeys the Meet Rule. Intermediate stages of the rewriting process do not maintain either well-formedness or type correctness according to either source-language or target-language rules, but the final result should be well-formed and type-correct according to target-language rules, and its execution behavior is intended to be identical to (and thereby *explain*) the execution behavior of the original source-language program.

We describe a series of rewriting steps as if each step were applied to all components and APIs before proceeding to the next step. However, up until the last step, "linking and final assembly" (see Section 5.11), each component may be processed entirely independently of all other components, relying only on information imported from APIs.

5.1 Create Object Constructors

If an object declaration with name U has a *ParameterList*

$$(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)$$

then copy each variable declaration $p_j\colon \mathrm{T}_j$ from the parameter list into the object declaration itself, adding the `var` keyword to each one that does not already have a `var` keyword. Then add to the object declaration a constructor definition of the form:

$$\begin{array}{l}\texttt{constructor}\ \mathrm{U}(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n) = \texttt{do}\\ \quad \texttt{self}.\mathit{field}\$p_1 := p_1\\ \quad \texttt{self}.\mathit{field}\$p_2 := p_2\\ \quad \ldots\\ \quad \texttt{self}.\mathit{field}\$p_n := p_n\\ \texttt{end}\end{array}$$

and delete the *ParameterList* and its surrounding parentheses from the object declaration. If the object declaration appears in a component, then add this top-level function definition to the component:

$$\mathrm{U}(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{U} = \texttt{new}\ \mathrm{U}(p_1, p_2, \ldots, p_n)$$

If an object declaration named U has no *ParameterList*, then it is a singleton object, implemented using the singleton pattern. Add to the object declaration this constructor:

`constructor` U() = `do end` ❀ It does nothing

and add to the component this top-level variable definition:

U: U = `new` U() ❀ The unique instance of U

5.2 Eliminate Fields

Getter and setter methods behave exactly like ordinary dotted methods; the only difference is that they are eligible to be invoked by field access expressions and field assignment expressions, respectively. Therefore all getters, setters, field accesses, and field assignments can be eliminated by rewriting them to dotted methods and dotted method invocations.

In the resulting target language code, field declarations appear only within objects, and field accesses and assignments occur only locally within the definition of the object (indeed, only within the one or two dotted methods associated with that field by the rewriting). This transformation is conventional; for lack of space, we do not present the details.

5.3 Eliminate Functional Methods

Now for a much meatier transformation: this step eliminates functional methods by rewriting them into a combination of top-level functions (that provide the appropriate invocation interface) and dotted methods (that provide the appropriate inheritance structure).

For every component C, for every trait or object declared in that component, let U be the declared name of the trait or object, and consider each functional method declaration or definition appearing within U. It will have the general form:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, \texttt{self}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

or:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, \texttt{self}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

where f is the name of the functional method; exactly one parameter position j in the declaration is occupied by the keyword `self`; $p_1, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n$ are names for its other parameters; $\mathrm{T}_1, \ldots, \mathrm{T}_{j-1}, \mathrm{T}_{j+1}, \ldots, \mathrm{T}_n$ are the respective types of those other parameters; and T is the result type. This functional method declaration or definition is replaced within U by a dotted method declaration or definition:

$f\$j(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

or:

$f\$j(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

Note that the name of this dotted method is $f\$j$ (thus the information about the position of the `self` parameter has not been lost), and that the `self` parameter has been eliminated (becoming instead the receiver for the dotted method). In addition, a new top-level function definition is added to component C:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, s\colon \mathrm{U}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = s.f\$j(p_1, p_2, \ldots, p_{j-1}, p_{j+1}, \ldots, p_n)$

where s is a freshly generated variable name appearing nowhere else in the program. In this manner, for any function call to a function named f for which the functional method was visible in the source language code, this newly created top-level function definition will be visible in the rewritten target language code, and will participate in overloading resolution with any other top-level functions named f within the same component.

Note that the top-level function definition created in this step will be further rewritten as described in Section 5.9, and the dotted method definition or declaration created in this step will be further rewritten as described in Section 5.10.

A related transformation is needed for functional method declarations appearing in an API. For every API A, for every trait declared in that API, let U be the declared name of the trait, and consider each functional method declaration or definition appearing within U; it will have the general form:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, \texttt{self}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

The rewriting deletes it from U and adds a top-level function declaration to API A:

$f(p_1\colon \mathrm{T}_1, \ldots, p_{j-1}\colon \mathrm{T}_{j-1}, s\colon \mathrm{U}, p_{j+1}\colon \mathrm{T}_{j+1}, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

where s is a freshly generated variable name appearing nowhere else in the program. In this manner, for any function call to a function named f for which the functional method has been imported from the API in the source language code, this newly created top-level function declaration will be imported instead in the rewritten target language code.

5.4 Rename Entities within Components

For every component C, for every top-level variable declaration or definition within C, change its name v to be $C\$\$v$, and change all references to v occurring within component C to be $C\$\$v$. (The double dollar sign avoids a naming conflict that might occur if C were the name "*field*". Mangling names consistently and unambiguously is tricky!)

For every component C, for every trait or object declaration within C, change its name U to be $C\$\mathrm{U}$, and change all references to U occurring within component C to be $C\$\mathrm{U}$.

In this way, entities declared with the same name in two different components are given distinct names.

Note that this step does *not* rename functions or methods. Necessary renaming of functions is taken care of in later steps (see Section 5.7 and Section 5.9).

5.5 Rewrite `import` and `export` Declarations

In every component and API, rewrite every `import` declaration with multiple import items into a series of `import` declarations each having a single import item. Next, rewrite each `import` declaration of the form: `import` $A.\{\, x \,\}$ into `import` $A.\{\, x \mapsto x \,\}$. These are, of course, merely "syntactic convenience" transformations.

Now, for every component C, process each resulting `import` declaration:

`import` $A.\{\, old \mapsto new \,\}$

(1) For every top-level function declaration in API A with the name old:

$old(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T}$

create in C a top-level function definition:

$new(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = \$A\$old(p_1, p_2, \ldots, p_n)$

(The leading dollar sign "$' indicates that the name $\$A\$old$ will need to be altered during the linking step described in Section 5.11.)

(2) For every trait declaration in API A with the name old, for throughout component C, change every reference to the trait new to $\$A\$old$. (After this point, names of traits and objects may be regarded as having global scope, transcending the boundaries of components and APIs.)

(3) For every `import` declaration in API A:

`import` $B.\{\ old' \mapsto new'\ \}$

such that $new' = old$, add to C a new `import` declaration:

`import` $B.\{\ old' \mapsto new\ \}$

and process it recursively.

After every `import` declaration in C has been processed (including the new ones generated in this step), delete every `import` declaration in component C.

5.6 Copy Inherited Methods and Assign Unique Integers

The scope of a top-level function declaration or definition is the entire component or API that contains it, while the scope of a local function declaration or definition is a block.

For the sake of this step, it is necessary to process the components and APIs in such an order that any given API is processed before any component that imports it. It is permitted for APIs to import each other, but overall, the trait and object declarations in a program must be processed in such an order that any given trait is processed before any trait or object that extends it. We also assume that as each method is copied, it somehow remains tagged with the name of the trait in which it originally appeared.

Within each component or API, for each maximal set of function declarations and definitions that have the same name and whose scopes are co-extensive, assign a distinct integer to each declaration or definition in the set. (The scoping rules of the source language follow those of Fortress [2], and so these maximal sets form a partition of the set of all function declarations and definitions.)

Within each API, for each trait declaration U, copy into U each dotted method declaration inherited (and not overridden) by U from its immediate supertraits. Then for each maximal set of dotted method definitions in U with the same name, assign a distinct integer to each definition in the set.

Within each component C, for each trait or object declaration U, there are three substeps:

(a) Consider each dotted method declaration or definition d inherited (and not overridden) by U from its immediate supertraits. If d is a declaration, say:

$m(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2 :, \ldots, p_n\colon \mathrm{T}_n)\colon T$

inherited from a trait V imported from an API A, and the integer assigned to d was i, then add to U a source-language dotted method definition with a target-language body that will not be further rewritten:

$m(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon T =$
`self` $.\$\$A\$V\$m\$i\$entry(p_1, p_2, \ldots, p_n)$

(The leading double dollar sign "$$' indicates that the name $\$\$A\$V\$m\$i\$entry$ will need to be altered during the linking step described in Section 5.11.)

(b) For each maximal set of dotted method definitions in U that have the same name, assign a distinct integer to each definition in the set.

(c) Revisit each declaration

$m(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon T =$
`self` $.\$\$A\$V\$m\$i\$entry(p_1, p_2, \ldots, p_n)$

introduced in substep (a), and let k be the integer that was assigned to it. Add to U a target-language dotted method definition that will not be further processed until the linking step described in Section 5.11:

$$\begin{aligned}\$A\$V\$m\$i\$dispatchHook(&\tau_0\colon \mathrm{Type},\\ &p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},\\ &p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},\\ &\ldots,\\ &p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =\end{aligned}$$
`self` $.C\$m\$k\$dispatchHook(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$

(This dotted method definition will overrride a definition provided by the component implementing API A.)

5.7 Rewrite Function Calls

Within every component C, consider every function call expression. If the call occurs within a trait or object declaration that has or inherits a definition or declaration of a dotted method with the same name, then the function call is rewritten into a dotted method call by adding `self.` to the front.

Otherwise, consider the statically most specific applicable function definition d, and let k be the integer that was assigned to that definition. Then there are two cases. If none of the argument expressions is an `asif` expression, then the function call has the form $f(a_1, a_2, \ldots, a_n)$. If d is a top-level definition, the function call is rewritten to:

$C\$f\$k\$entry(a_1, a_2, \ldots, a_n)$

(Note the incorporation of the name C of the containing component.) Otherwise it is rewritten to:

$f\$k\$entry(a_1, a_2, \ldots, a_n)$

In the other case, at least one of the arguments is an `asif` expression, for example:

$f(a_1, a_2$ `asif` T_2, a_3 `asif` $\mathrm{T}_3, \ldots, a_n)$

Such a function call is rewritten to:

```
do
  (z_1, z_2, z_3, ..., z_n) = (a_1, a_2, a_3, ..., a_n)
  w(z_1, z_1.ilk(), z_2, T_2.type, z_3, T_3.type, ..., z_n, z_n.ilk())
end
```

where the name w is actually $C\$f\$k\$dispatch$ (note the incorporation of the name C of the containing component) if d is a top-level definition, and otherwise is simply $f\$k\$dispatch$, and $z, z_1, z_2, \ldots, z_n$ are freshly generated variable names that occur nowhere else in the program.

Note that it would actually be correct always to use this second rewriting rule, even for calls that do not contain an `asif` expression. However, having no `asif` expression is the common case, and we choose to treat it specially. This structure also provides more flexibility for optimization (see Section 6.1).

5.8 Rewrite Dotted Method Calls

This section closely parallels the previous section.

For every component C, consider each dotted method call expression that occurs within it. Consider the statically most specific applicable dotted method definition d for that call, and let k be the integer that was assigned to that definition. Then there are two cases. If the receiver expression is not an `asif` expression, then none of the argument expressions can be an `asif` expression, so the dotted method call has the form $x.m(a_1, a_2, \ldots, a_n)$, and it is rewritten to:

$x.C\$m\$k\$entry(a_1, a_2, \ldots, a_n)$

In the other case, the receiver expression is an `asif` expression, and one or more arguments may also be `asif` expressions, for example:

$(y \texttt{ asif } T).m(a_1, a_2 \texttt{ asif } \mathrm{T}_2, a_3 \texttt{ asif } \mathrm{T}_3, \ldots, a_n)$

Such a dotted method call is rewritten to:

```
do
  (z, z1, z2, z3, ..., zn) = (y, a1, a2, a3, ..., an)
  z.C$m$k$dispatch(T.type, z1, z1.ilk(), z2, T2.type,
                   z3, T3.type, ..., zn, zn.ilk())
end
```

where $z, z_1, z_2, \ldots, z_n$ are freshly generated variable names that occur nowhere else in the program.

5.9 Rewrite Overloaded Functions

In every component and API, for every function definition d, let S_f be the maximal set of function declarations and definitions that have the same name f and whose scopes are co-extensive, let the integer that was assigned to d (see Section 5.6) be k, and suppose that d has n parameters:

$f(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2 :, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

Let S_f^k be the subset of S_f consisting of all definitions in S_f that are strictly more specific than definition d. Let U_f^k be the "upper fringe" of S_f^k, that is, the set of all definitions d' in S_f^k such that there is no definition d'' in S_f^k such that d' is strictly more specific than d''. Now let D_f^k be a "dispatch set," that is, any arbitrarily chosen set of function definitions such that $U_f^k \subseteq D_f^k \subseteq S_f^k$. Let q be the size of this dispatch set, and let $k_1, k_2, \ldots, k_q$ be the integers assigned to its elements (which may be ordered arbitrarily for the purpose of assigning indices $1, 2, \ldots, q$ to their respective integers). Then rewrite definition d into three functions in the same scope as follows:

$C\$f\$k\$entry(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} =$
$\quad C\$f\$k\$dispatch\big(p_1, p_1.ilk(), p_2, p_2.ilk(), \ldots, p_n, p_n.ilk()\big)$

$C\$f\$k\$dispatch(p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},$
$\qquad\qquad p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},$
$\qquad\qquad \ldots,$
$\qquad\qquad p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =$
$\quad$ `if` $C\$f\$k_1\$applicable(\tau_1, \tau_2, \ldots, \tau_n)$ `then`
$\quad\quad C\$f\$k_1\$dispatch(p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$
$\quad$ `elif` $C\$f\$k_2\$applicable(\tau_1, \tau_2, \ldots, \tau_n)$ `then`
$\quad\quad C\$f\$k_2\$dispatch(p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$
$\quad \ldots$
$\quad$ `elif` $C\$f\$k_q\$applicable(\tau_1, \tau_2, \ldots, \tau_n)$ `then`
$\quad\quad C\$f\$k_q\$dispatch(p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$
$\quad$ `else` $C\$f\$k(p_1, p_2, \ldots, p_n)$ `end`

$C\$f\$k(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

where $\tau_1, \tau_2, \ldots, \tau_n$ are freshly generated variable names that occur nowhere else in the program. (A separate function definition $C\$f\$k$, identical to the original except its name, is retained, even though it is called from only one place, so that the expression e will not be duplicated if the function $C\$f\$k\$dispatch$ is inlined for optimization purposes; see Section 6.1.) However, if the dispatch set D_f^k is empty, then the second function is simply:

$C\$f\$k\$dispatch(p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},$
$\qquad\qquad p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},$
$\qquad\qquad \ldots,$
$\qquad\qquad p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =$
$\quad C\$f\$k(p_1, p_2, \ldots, p_n)$

In addition, if there is at least one definition d' in S_f such that d is more specific than d', then create a fourth function in the same scope:

$C\$f\$k\$applicable(\tau_1\colon \mathrm{Type}, \tau_2\colon \mathrm{Type}, \ldots, \tau_n\colon \mathrm{Type})\colon \mathrm{Boolean} =$
$\quad \tau_1.isSubtype(\mathrm{T}_1.\texttt{type})$ `&&`
$\quad \tau_2.isSubtype(\mathrm{T}_2.\texttt{type})$ `&&`
$\quad \ldots$ `&&`
$\quad \tau_n.isSubtype(\mathrm{T}_n.\texttt{type})$

5.10 Rewrite Overloaded Dotted Methods

This section is more complex than the previous section.

In every component, for every object declaration U, for every dotted method definition d in U, let S_m be the maximal set of dotted method definitions in U that have the same name m, let the integer that was assigned to d (see Section 5.6) be k, and suppose that d has n parameters:

$m(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2 :, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$

Let S_m^k be the subset of S_m consisting of all definitions in S_m that are strictly more specific than definition d. Let U_m^k be the "upper fringe" of S_m^k, that is, the set of all definitions d' in S_m^k such that there is no definition d'' in S_m^k such that d' is strictly more specific than d''. Now let D_m^k be a "dispatch set," that is, any arbitrarily chosen set of dotted method definitions such that $U_m^k \subseteq D_m^k \subseteq S_m^k$. Let q be the size of this dispatch set, and let $k_1, k_2, \ldots, k_q$ be the integers assigned to its elements (which may be ordered arbitrarily for the purpose of assigning indices $1, 2, \ldots, q$ to their respective integers). Then rewrite definition d into three

dotted method definitions within the same trait or object declaration as follows:

$$C\$U\$m\$k\$entry(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} =$$
$$\quad C\$U\$m\$k\$dispatch\big(\texttt{self}.ilk(),$$
$$\qquad\qquad p_1, p_1.ilk(), p_2, p_2.ilk(), \ldots, p_n, p_n.ilk()\big)$$

$$C\$U\$m\$k\$dispatchHook(\tau_0\colon \mathrm{Type},$$
$$\qquad\qquad p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},$$
$$\qquad\qquad p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},$$
$$\qquad\qquad \ldots,$$
$$\qquad\qquad p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =$$
$$\quad C\$U\$m\$k\$dispatch\big(\texttt{self}.ilk(),$$
$$\qquad\qquad p_1, p_1.ilk(), p_2, p_2.ilk(), \ldots, p_n, p_n.ilk()\big)$$

$$C\$U\$m\$k\$dispatch(\tau_0\colon \mathrm{Type},$$
$$\qquad\qquad p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},$$
$$\qquad\qquad p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},$$
$$\qquad\qquad \ldots,$$
$$\qquad\qquad p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =$$
$$\quad \texttt{if self.}C\$U\$m\$k_1\$applicable(\tau_0, \tau_1, \tau_2, \ldots, \tau_n) \texttt{ then}$$
$$\quad\quad \texttt{self.}C\$U\$m\$k_1\$dispatchHook(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$$
$$\quad \texttt{elif self.}C\$U\$m\$k_2\$applicable(\tau_0, \tau_1, \tau_2, \ldots, \tau_n) \texttt{ then}$$
$$\quad\quad \texttt{self.}C\$U\$m\$k_2\$dispatchHook(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$$
$$\quad \ldots$$
$$\quad \texttt{elif self.}C\$U\$m\$k_q\$applicable(\tau_0, \tau_1, \tau_2, \ldots, \tau_n) \texttt{ then}$$
$$\quad\quad \texttt{self.}C\$U\$m\$k_q\$dispatchHook(\tau_0, p_1, \tau_1, p_2, \tau_2, \ldots, p_n, \tau_n)$$
$$\quad \texttt{else self.}C\$U\$m\$k(p_1, p_2, \ldots, p_n) \texttt{ end}$$

$$C\$U\$m\$k(p_1\colon \mathrm{T}_1, p_2\colon \mathrm{T}_2, \ldots, p_n\colon \mathrm{T}_n)\colon \mathrm{T} = e$$

where $\tau_0, \tau_1, \tau_2, \ldots, \tau_n$ are freshly generated variable names that occur nowhere else in the program. (A separate method definition $C\$U\$m\$k\$$, identical to the original except for its name, is retained, even though it is called from only one place, so that the expression e will not be duplicated if the method $C\$U\$m\$k\$dispatch$ is inlined for optimization purposes; see Section 6.1.) However, if the dispatch set D^k_m is empty, then the third method is simply:

$$C\$U\$m\$k\$dispatch(\tau_0\colon \mathrm{Type},$$
$$\qquad\qquad p_1\colon \mathrm{T}_1, \tau_1\colon \mathrm{Type},$$
$$\qquad\qquad p_2\colon \mathrm{T}_2, \tau_2\colon \mathrm{Type},$$
$$\qquad\qquad \ldots,$$
$$\qquad\qquad p_n\colon \mathrm{T}_n, \tau_n\colon \mathrm{Type})\colon \mathrm{T} =$$
$$\quad \texttt{self.}C\$U\$m\$k(p_1, p_2, \ldots, p_n)$$

In addition, if there is at least one definition d' in S_m such that d is more specific than d', then create a fourth dotted method in the same trait or object declaration:

$$C\$U\$m\$k\$applicable(\tau_0\colon \mathrm{Type}, \tau_1\colon \mathrm{Type}, \tau_2\colon \mathrm{Type}, \ldots,$$
$$\qquad\qquad \tau_n\colon \mathrm{Type})\colon \mathrm{Boolean} =$$
$$\quad \tau_0.isSubtype(\mathrm{W}.\texttt{type}) \texttt{ \&\&}$$
$$\quad \tau_1.isSubtype(\mathrm{T}_1.\texttt{type}) \texttt{ \&\&}$$
$$\quad \tau_2.isSubtype(\mathrm{T}_2.\texttt{type}) \texttt{ \&\&}$$
$$\quad \ldots \texttt{ \&\&}$$
$$\quad \tau_n.isSubtype(\mathrm{T}_n.\texttt{type})$$

where W is the name of the trait or object in which the definition d originally appeared.

5.11 Linking and Final Assembly

(Up to this point, all transformations on an API or component have required only information in that API or component plus information imported from APIs, and therefore correspond to a process of separate compilation of each component. This step corresponds to a linking step that binds multiple components into a single program by renaming certain entities and copying inherited methods. In practice, within a virtual machine implementation, one would simply copy method pointers rather than duplicating code.)

For every component C, for every name within C of the form "$\$A\$\ldots$", identify the *implementing component* D for A (the (necessarily unique) component that contains the statement `export` A), and alter the name "$\$A\$\ldots$" to be "$D\$\ldots$" instead; and for every name within C of the form "$\$\$A\$V\$m\$i\$\ldots$", identify the implementing component D for A, then identify the method declaration for m within trait V within API A that has integer i assigned to it, then identify the method definition for m within trait V within component D that has the same signature, and let k be the integer assigned to it, then alter the name to be "$D\$m\$k\$\ldots$" instead.

All APIs may now be discarded.

For every object declaration that appears in a component, add into the object declaration a copy of every dotted method definition that it inherits and is not overridden, even if that inherited definition is in another component; note that overriding should occur only for methods whose names end in "$\$dispatchHook$". Change the keyword `object` to `class`.

Now, for every trait declaration that appears in a component, delete the body of every dotted method definition in the trait, thus converting it to an abstract dotted method declaration. Note that the trait declaration now contains no executable code, only method declarations. Change the keyword `trait` to `interface`.

Once this has been done for all components in the program, delete every `export` declaration in every component.

The final program in the target language consists of the concatenation of the residual contents of all components, with the leading "`component` *ComponentName*" and trailing "`end`" of each component removed.

6. Variations and Optimizations

The rewriting process described in Section 5 is intended to be general and reasonably easy to understand. In practice some useful variations and optimizations are available for use when appropriate.

6.1 Inlining

The intermediary *dispatch* and *applicable* functions and methods introduced by the rewritings described in Section 5.9 and Section 5.10 may often be profitably inlined. To work a specific example, consider these definitions for an overloaded local function named *which*, with their assigned integers shown in comments:

$$which(x\colon \mathrm{Object}, y\colon \mathrm{Object})\colon \mathrm{String} = \text{“}\texttt{neither}\text{”} \quad ❀\ 1$$
$$which(x\colon \mathrm{Object}, y\colon \mathrm{String})\colon \mathrm{String} = \text{“}\texttt{second}\text{”} \quad ❀\ 2$$
$$which(x\colon \mathrm{String}, y\colon \mathrm{Object})\colon \mathrm{String} = \text{“}\texttt{first}\text{”} \quad ❀\ 3$$
$$which(x\colon \mathrm{String}, y\colon \mathrm{String})\colon \mathrm{String} = \text{“}\texttt{both}\text{”} \quad ❀\ 4$$

The process described in Section 5.9 will (assuming a minimal choice of dispatch set) rewrite the first definition to:

```
which$1$entry(x: Object, y: Object): String =
  which$1$dispatch(x, x.ilk(), y, y.ilk())

which$1$dispatch(x: Object, τ1: Type,
                 y: Object, τ2: Type): String =
  if which$2$applicable(τ1, τ2) then
    which$2$dispatch(x, τ1, y, τ2)
  elif which$3$applicable(τ1, τ2) then
    which$3$dispatch(x, τ1, y, τ2)
  else which$1(x, y) end

which$1$applicable(τ1: Type, τ2: Type): Boolean =
  τ1.isSubtype(Object.type) && τ2.isSubtype(Object.type)

which$1(x: Object, y: Object): String = "neither"
```

and similarly for the other three definitions. If we assume that an optimizer can rewrite $x.ilk().isSubtype(\mathrm{T}.\texttt{type})$ to simply x `instanceof` T, then after inlining and elimination of redundant computations, one may obtain:

```
which$1$entry(x: Object, y: Object): String =
  if y instanceof String then
    if x instanceof String then which$4(x, y)
    else which$2(x, y) end
  elif x instanceof String then which$3(x, y)
  else which$1(x, y) end
```

which is a not unreasonable implementation of the dispatch process as an executable discrimination tree. This is exactly the sort of thing one would expect from, say, a competent Java compiler [16]. Further transformations are of course possible, perhaps based on dynamic profiling information.

6.2 Exploiting Single Inheritance

The rewriting step described in Section 5.11 copies *all* inherited dotted method definitions from traits down into objects. If the target environment for program execution supports single inheritance, then it may be advantageous to exploit it by choosing a subset of the edges of the type hierarchy so as to form a tree providing exactly one path from every object type to the root of the type hierarchy. Every Fortress trait can then be represented by a Java interface and an associated Java class that extends it. (Note that the Fortress trait named Object is *not* represented by the Java class `Object`, but rather by a Java interface named something like `Fortress$Object`, which in turn extends a Java interface named something like `Fortress$Any`.) The Fortress type hierarchy is then represented by `extends` relationships among these Java interfaces. The Java class associated with a Fortress object type is used to hold the (rewritten) dotted method definitions copied down from supertraits that are *not* accessible through the chosen tree; this class also extends the Java class associated with the Fortress supertrait that *is* accessible through the chosen tree, and therefore inherits dotted methods from it.

```
component Lib
export Library
p(x: Object): String = "/" x "/"
p(x: ℤ): String = "#" x "#"
end Lib

api Library
p(x: Object): String
end Library

component User
import Library.{ p }
export Executable
object Foo(x: String)
  p(self): String = "<" x ">"
end
p(x: ℤ): String = "[" x "]"
run(): () = do
  q(z: Object): String = p(z)
  println(p(Foo("hello")) " versus " q(Foo("hello")))
  println(p(17) " versus " q(17))
  println(p(6.375) " versus " q(6.375))
end
end User
```

Figure 3. Example code in source language

If this representation strategy is used, then the rewriting described in Section 5.11 (delete the body of every dotted method definition in the trait) should not be used.

7. Big Example

Two example components and an example API are shown in Figure 3. For the purposes of this example we extend the source and target languages to include string concatenation through expression juxtaposition, rather than translating such operations into method calls. When the run function is invoked, the program prints:

```
<hello> versus <hello>
[17] versus [17]
/6.375/ versus /6.375/
```

One point of this example is that the printed results are the same when p is called either directly or (through q) indirectly, even though the dispatch is achieved statically when p is called directly but dynamically when called through q (because the parameter z of function q has type Object).

Another point is that the example contains two definitions of p with parameter type $\mathbb{Z}$. If the underlying implementation were to lump all the definitions of p into a single CLOS-style "generic function" then the definitions would not form a meet-bounded lattice and the call $p(17)$ would be ambiguous; but the fact that two definitions are in a separate component Lib and only one is exported means that only three definitions are visible within component User, and they do form a meet-bounded lattice.

```
component Lib
export Library
p(x: Object): String = "/" x "/"                                    ✻ 1
p(x: ℤ): String = "#" x "#"                                         ✻ 2
end Lib

component User
export Executable
p(x: Object) = $Library$p(x)                                        ✻ 1
class User$Foo
  var field$x: String
  constructor Foo(x: String) = do self.field$x := x end
  p$1(): String = "<" self.field$x ">"                              ✻ 1
end
Foo(x: String): User$Foo = new User$Foo(x)                          ✻ 1
p(s: Foo): String = s.p$1()                                         ✻ 2
p(x: ℤ): String = "[" x "]"                                         ✻ 3
run(): () = do                                                      ✻ 1
  q(z: Object): String = p(z)                                       ✻ 1
  println(p(Foo("hello")) " versus " q(Foo("hello")))
  println(p(17) " versus " q(17))
  println(p(6.375) " versus " q(6.375))
end
end User
```

Figure 4. Example code, partially rewritten

Space does not permit exhibiting snapshots of the code after each rewriting step. Figure 4 shows what the code looks like after integers have been assigned to function and method definitions (see Section 5.6). Figure 5 shows the final result, after all rewriting steps plus a certain amount of procedure inlining to reduce clutter. Of particular interest are the two functions *Lib*$*p*$1$*entry* and *User*$*p*1*entry* (into which the corresponding dispatch functions have been inlined).

Note that if the component User did not contain the definition:

$p(x: \mathbb{Z}): \text{String} = \text{"[" } x \text{ "]"}$

then the program would still be correct, and would instead print:

```
<hello> versus <hello>
#17# versus #17#
/6.375/ versus /6.375/
```

For argument 17 (of type $\mathbb{Z}$), the static or dynamic dispatch within component User would transfer control to the first definition of p in component Lib, which would then dynamically dispatch to the second definition in component Lib. While this second definition is not imported into component User and therefore is not directly visible, it does form a part of the *implementation* of the overloaded function p within component Lib.

```
Lib$p$1$entry(x: Object): String =
  if x instanceof ℤ then Lib$p$2(x)
  else Lib$p$1(x) end
Lib$p$1(x: Object): String = "/" x "/"
Lib$p$2(x: ℤ): String = "#" x "#"

User$p$1$entry(x: Object) =
  if x instanceof User$Foo then User$p$2(x)
  elif x instanceof ℤ then User$p$3(x)
  else Lib$p$1$entry(x) end
class User$Foo
  var field$x: String
  constructor Foo(x: String) = do self.field$x := x end
  p$1$1(): String = "<" self.field$x ">"
end
User$Foo$1(x: String) = new User$Foo(x)
User$p$2(s: User$Foo): String = s.p$1$1()
User$p$3(x: ℤ): String = "[" x "]"
User$run$1(): () = do
  q$1(z: Object): String = User$p$1$entry(z)
  println( User$p$2( User$Foo$1("hello")) " versus "
           q$1( User$Foo$1("hello")))
  println( User$p$3(17) " versus " q$1(17))
  println( Lib$p$1$entry(17) " versus " q$1(6.375))
end
```

Figure 5. Example code, completely rewritten

8. Extension to Parametrically Polymorphic Functions and Methods

Now suppose that we wish for our source language to include functions and methods with explicit static type parameters. We might, for example, write a function *both* that takes two values of type T and a predicate on values of type T, and returns *true* iff the predicate is true of both values:

$both[\![T]\!](a: \mathrm{T}, b: \mathrm{T}, p: \mathrm{T} \to \text{Boolean}): \text{Boolean} = p(a) \wedge p(b)$

Determining whether sets of such polymorphic functions form valid overloadings is an interesting problem that is beyond the scope of this paper but is addressed by a companion paper [1]. But once an overload set has passed the static type-checking process, we know that the signatures of the functions form a meet-bounded lattice (according to the rule that function f_1 is strictly more specific than function f_2 if and only if f_2 is applicable to every set of arguments to which f_1 is applicable and there is some set of arguments to which f_2 is applicable and f_1 is not). Therefore exactly the same rewriting schema described in Section 5.9 applies, and similarly for dotted methods. All that is necessary is to provide an appropriate implementation of the *applicable* functions or methods. In this example, *both*1*applicable* needs to extract the respective ilks A, B, and P from the arguments a, b, and p, verify that the ilk of p is an arrow type, and extract the argument type C from that arrow type. If this definition of *both* is to be applicable, the type T must be at least $A \cup B$, and can be at most C. The necessary test is therefore $(A \cup B).isSubtype(C)$, which can be further simplified to $A.isSubtype(C) \wedge B.isSubtype(C)$. Putting it all together, we have:

```
both$1$applicable(τ1: Type, τ2: Type, τ3: Type): Boolean =
  if τ3 instanceof ArrowType then
    τ4 = τ3.argumentType()
    (τ1.isSubtype(τ4) && τ2.isSubtype(τ4) &&
     τ3.resultType().isSubtype(Boolean.type))
  else false end
```

If this example were modified to bound the type parameter:

```
both⟦T extends ℚ⟧(a: T, b: T, p: T → Boolean): Boolean =
  p(a) ∧ p(b)
```

then one might imagine that one must first find an actual type X for T that is a subtype of ℚ, and then decide whether the ilks of a, b, and p are subtypes of X, X, and X → Boolean, respectively. But in fact it suffices to test the extracted ilks from covariant positions against the bound:

```
both$1$applicable(τ1: Type, τ2: Type, τ3: Type): Boolean =
  if τ3 instanceof ArrowType then
    τ4 = τ3.argumentType()
    (τ1.isSubtype(ℚ.type) && τ1.isSubtype(τ4) &&
     τ2.isSubtype(ℚ.type) && τ2.isSubtype(τ4) &&
     τ3.resultType().isSubtype(Boolean.type))
  else false end
```

In general, determining applicability involves deciding whether a set of type constraints is satisfiable. But the testing of applicability at run time is less complicated than one might think when one considers that (a) the methods have already been statically checked for type correctness, and (b) it is not necessary to explicitly construct or identify types for the type parameters, but only to prove their existence.

9. Related Work

We have already made comparisons between our work and features of CLOS [13, 4, 22, 14], Java [15], MultiJava [10, 11], Scala [20], and especially the λ&-calculus of Castagna, Ghelli, and Longo [6], which is earliest work we know of to recommend symmetric multimethod dispatch as a central feature of an object-oriented language design.

Of all the related languages we have surveyed, however, the one closest in design and intent to what we present here is Dubious [18], which provides symmetric dynamic multimethod dispatch while allowing a program to be divided into modules that can be separately type-checked statically. This work differs from Dubious in these respects: Dubious is a classless (prototype-oriented) object system, whereas Fortress traits are classes in this sense; Dubious has only explicitly declared objects, whereas this work supports dynamically created (allocated) objects and state; and importing a Dubious module is an all-or-nothing proposition (though the cited paper does sketch a possible way to introduce a `private` keyword to shield some objects in a module), whereas Fortress APIs and import declarations allow fine-grained selective export and import of only parts of a component—in particular, it is possible to export only selected methods of a trait, rather than all methods. Two other languages from the same research group that are also closely related to the our work are Cecil [7, 8, 9] and EML [17].

10. Conclusions and Future Work

Namespace control in object-oriented languages is tricky. On the one hand, there is a desire to inherit method definitions implicitly and to be able to override and overload them. On the other hand, there is a desire to be able to control access to specific methods by controlling their names. On the third hand, if a trait T is made public but a method m is kept private, then third parties extending T should not be prevented from using the name m as a method name, and the two methods (or sets of methods) should not interact. The framework we present solves this problem effectively.

We agree with Millstein and Chambers that symmetric multimethod dispatch is far preferable to other disciplines: "[T]he symmetric multimethod dispatching semantics … is more natural and less error-prone, since it reports potential ambiguities rather than silently resolving them" [18, §4.1] Moreover, the Meet Rule has beneficial consequences not only for aiding programmer understanding but for enabling separate compilation and a distributed (rather than monolithic) implementation of multimethod dispatch that is amenable to optimization.

Functional methods are an effective approach to solving the binary method problem. The advantage over dotted methods is that any argument position may serve as the receiver; the advantage over ordinary multimethods (functions) is that a trait may declare an abstract functional method, thereby obligating any object that extends the trait to implement it. We have shown how to implement this feature in terms of functions and dotted methods in a manner that interacts well with namespace control.

In the future we hope to extend this framework to parametrically polymorphic traits and objects.

References

[1] Type-checking modular multiple dispatch with parametric polymorphism and multiple inheritance. Submitted to OOPSLA 2011.

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., S. Tobin-Hochstadt, J. Dias, C. Eastlund, C. Flood, Y. Lev, C. McCosh, J. D. Nielsen, and D. Smith. The Fortress Language Specification, version 1.0. Technical report, Sun Microsystems Laboratories, Burlington, MA, Mar. 2008.

[3] E. Allen, J. J. Hallett, V. Luchangco, S. Ryu, and G. Steele. Modular multiple dispatch with multiple inheritance. In *SAC '07: Proc. 2007 ACM Symposium on Applied Computing*, pages 1117–1121, New York, NY, USA, 2007. ACM.

[4] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification. *SIGPLAN Notices*, 23(special issue):1–142, 1988.

[5] K. Bruce, L. Cardelli, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[6] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *LFP '92: Proc. 1992 ACM Conference on LISP and Functional Programming*, pages

182–192, New York, NY, USA, 1992. ACM.

[7] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56, London, UK, 1992. Springer-Verlag.

[8] C. Chambers. The Cecil language: Specification and rationale. Technical Report UW-CSE-93-03-05, University of Washington, Seattle, WA, Mar. 1993.

[9] C. Chambers et al. The Cecil language: Specification and rationale. Technical report, University of Washington, Seattle, WA, Feb. 2004.

[10] C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular open classes and symmetric multiple dispatch for Java. In *OOPSLA '00: Proc. 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 130–145, New York, NY, USA, 2000. ACM.

[11] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[12] ECMA. *C# Language Specification*, Dec. 2001. Standard ECMA-334.

[13] R. P. Gabriel and L. DeMichiel. An overview of the Common Lisp Object System. In *Proceedings of the European Conference on Object-Oriented Programming ECOOP 1987*, pages 151–170, Paris, July 1987. Proceedings published by Springer-Verlag as LNCS 276.

[14] R. P. Gabriel, J. L. White, and D. G. Bobrow. CLOS: Integrating object-oriented and functional programming. *Comm. ACM*, 34(9):29–38, 1991.

[15] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, Massachusetts, 1996.

[16] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™client compiler for Java 6. *ACM Trans. Arch. Code Optim.*, 5(1):1–32, 2008.

[17] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Trans. Program. Lang. Syst.*, 26(5):836–889, 2004.

[18] T. Millstein and C. Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, 2002.

[19] E. R. Murphy-Hill and A. P. Black. Traits: Experience with a language feature. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 275–282, New York, NY, USA, 2004. ACM Press.

[20] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA '05: Proc. 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–57, New York, NY, USA, 2005. ACM.

[21] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings of the 17th European Conference on Object-Oriented Programming*. Springer, July 2003.

[22] G. L. Steele Jr., S. E. Fahlman, R. P. Gabriel, D. A. Moon, D. L. Weinreb, D. G. Bobrow, L. G. DeMichiel, S. E. Keene, G. Kiczales, C. Perdue, K. M. Pitman, R. C. Waters, and J. L. White. *Common Lisp: The Language (Second Edition)*. Digital Press, Bedford, MA, 1990.

[23] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.