Fine-Grained Function Visibility for Multiple Dispatch with Multiple Inheritance

Jieung Kim[1], Sukyoung Ryu[2], Victor Luchangco[3], and Guy L. Steele Jr.[3]

[1] Department of Computer Science, Yale University
[2] Department of Computer Science, KAIST
[3] Oracle Labs

Abstract. Object-oriented languages with multiple dispatch and multiple inheritance provide rich expressiveness but statically and modularly checking programs in such languages to guarantee that no ambiguous calls can occur at run time has been a difficult problem. We present a core calculus for Fortress, which provides various language features—notably *functional methods* and *components*—and solves the problem. Functional methods are declared within traits and may be inherited but are invoked by ordinary function calls, and therefore compete in overloading resolution with ordinary function declarations. A novel component system governs "visibility" of types and functions, and allows fine-grained control over the import of overloaded functions. We formally define the type system of Fortress with a set of static rules to guarantee no ambiguous calls at run time, and mechanize the calculus and its type safety proof in Coq.

1 Introduction

A longstanding problem for systems that support multiple inheritance is what to do when a method is invoked on an object that inherits that method from multiple parents: which inherited method should be executed? More generally, when a method or a function (collectively called *functional*) is *overloaded*—that is, there are multiple declarations of the same name—which declaration should be used when the functional is invoked? Intuitively, we want to use the most specific declaration that is applicable to the call. But there might not be a unique most-specific declaration: There may be two (or more) applicable declarations that are more specific than any of the other declarations but incomparable with each other. In this case, we say the invocation is *ambiguous*. Guaranteeing that there will be no ambiguous calls at run time is difficult in object-oriented languages because some declarations that are applicable at run time might not be applicable statically.

Castagna *et al.* [4] address this problem by requiring that the signatures (i.e., the parameter types) of overloaded functional declarations form a meet-bounded lattice. This approach has been taken by several languages that support multiple dispatch and multiple inheritance, such as MultiJava [7], Cecil [5], and Dubious [14]. We have followed this approach in the Fortress programming language [1], in which this requirement is called the *Meet Rule*, and our experience is that it feels natural in practice, and that checking it statically helps expose programming errors.

C.-c. Shan (Ed.): APLAS 2013, LNCS 8301, pp. 156–171, 2013.

However, checking this condition statically is complicated by modularity: Fortress programs are partitioned into modules, each of which must import functionality from other modules that it wants to use. Fortress allows fine-grained control over the import of not only type declarations but also overloaded functional declarations, so that different modules may see different sets of declarations for a given functional. Thus, calls with the exact same arguments but from different components to dispatch to different functionals. In contrast, prior languages avoid this complication by sacrificing fine-control over importing functional declarations: in these languages, either all or none of the overloaded declarations of a functional is visible in any given program scope [2,3,5,7,11].

Fortress provides a novel solution to the "operator method problem": *functional methods* [2]. A functional method may designate any argument, not just the textually leftmost, to be treated as the "dispatch target" or "receiver" so that it can enjoy a function call syntax. Functional methods are inherited like conventional dotted methods but have the visibility of top-level functions (i.e., they are in the top-level scope of a module). Thus, they can (and must) be imported to be used in a different module, giving programmers the same fine-grained control for functional methods as for top-level functions. (For this reason, we often find ourselves preferring functional methods to dotted methods in Fortress.) However, the expressive power of functional methods does not come for free. Allowing a functional method to designate any argument as the receiver enlarges the set of overloaded declarations among which the most specific one is chosen. Selectively importing operator (functional method) declarations requires that functional methods be overloaded with top-level functions, which adds complexity to the static checks to guarantee the existence of the single most specific one among them.

In this paper, we extend the state of the art in statically typed object-oriented languages with symmetric multiple dispatch and multiple inheritance by allowing each set of overloaded functional declarations to have its own visibility via fine-grained imports. We present a core calculus of Fortress in which a program can be divided into components that can be modularly type-checked, *such that the components provide complete namespace control over all top-level names*; names of not only types, but also overloaded functions and functional methods, may be selectively imported. The Meet Rule makes possible a modular static checking of multimethod dispatch that enables separate compilation. We used Coq [8] to mechanize the calculus to prove the soundness of its type system, guaranteeing that there are no ambiguous calls at run time. Our Coq mechanization of the calculus and type soundness proof is publicly available [10].

2 Fortress Language Features

In this section, we describe the language features of Fortress relevant to overloading, dispatch and fine-grained namespace control, and the rules that enable modular type checking.

2.1 Traits and Objects for Multiple Inheritance

Fortress organizes objects into a multiple-inheritance hierarchy based on *traits* [19]. It may be helpful to think of a Fortress trait as a Java interface that can also contain

concrete method declarations, 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 method declarations, and may inherit method declarations from multiple supertraits (traits that they *extend*). 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 Sather [22]. By separating types into traits and objects, Fortress supports multiple inheritance without suffering from conflicts between inherited fields. As in Java and similar statically typed object-oriented languages, the name of a trait or object serves as the name of a type, which is the set of all objects at or below the named position in the trait hierarchy.

2.2 Three Kinds of Functionals for Multiple Dispatch

We consider three kinds of *functionals* in Fortress: *i)* traditional dotted methods associated with objects, *ii)* top-level functions not associated with any objects as in C++ [21], and *iii)* functional methods. As in Java, dotted methods are invoked with a *dotted method call* of the form "$e.m(e_1, \ldots, e_n)$" while top-level functions and functional methods are invoked with a *functional call* of the form "$m(e_1, \ldots, e_n)$" without any dot. Dotted methods and functional methods are collectively called *methods*; top-level functions and functional methods are collectively called *functions*.

Functionals may be *overloaded*; that is, several methods within an object, and several functions declared in the same scope, may have the same name. This raises the issue of overload resolution: at run time, we must resolve the overloading to determine what code to execute for each functional call. Typically, overloading is resolved by *dispatching* to the *most specific* functional declaration from among those declarations that are accessible and applicable, where declarations are compared by their parameter types. With *symmetric multiple dispatch*, the types of all the parameters of a functional declaration are considered equally in this comparison.

In an early report [2], we considered a restriction of Fortress in which top-level functions and functional methods could not be overloaded. Removing this restriction introduced some new issues, which we address in this paper. (Fortress does not allow dotted methods to be overloaded with functional methods or top-level functions, so we do not consider this case in this paper. However, if, within the body of a trait or object, dotted methods could be invoked by functional calls and overloaded with top-level functions and/or functional methods, we can handle this case by considering every dotted method declaration to also declare a function whose parameters do not include the receiver, and include this function in the set of candidates that must satisfy the overloading rules described in Section 2.4.)

2.3 Components and Selective Imports for Modularity

A program can be divided into *modules* that can be compiled separately, and provide a form of namespace control. In Fortress, these modules are called *components*, which can contain declarations of top-level functions, traits, and objects; trait and object declarations may contain dotted methods and functional methods. A component may selectively import type names and function names from other components. One of the main

```
component IntegerToString
  trait Int
    asString() = "-" ‖ (− self).asString()
  end
  trait Nat extends Int
    asString() =
      if self < 10
      then "0123456789"[self : self]
      else
        q = self ÷ 10
        r = self − 10 q
        q.asString() ‖ r.asString()
      end
  end
end IntegerToString
```

```
component IntegerToStringFunction
  trait Int end
  trait Nat extends Int end
  asString(x: Int) =
    "-" ‖ asString(−x)
  asString(x: Nat) =
    if x < 10
    then "0123456789"[x : x]
    else
      q = x ÷ 10
      r = x − 10 q
      asString(q) ‖ asString(r)
    end
end IntegerToStringFunction
```

Fig. 1. Overloaded dotted methods (left) and top-level functions (right)

contributions of this paper is to explain how the overloading checks and the overloading resolution safely interact with the namespace control imposed by components.

Let Int and Nat name the types whose members consist of the integers and natural numbers (i.e., nonnegative integers) respectively. Thus, Nat is a subtype of Int. Then they may be implemented in the component IntegerToString as illustrated in the left side of Figure 1 (shown only in part; they might define other methods) where ‖ denotes string concatenation, `self` denotes the receiver object (like `this` in Java), and "0123456789"[self : self] denotes indexing of the string, a linear sequence of characters, with a range of size 1. Note the members of Int are the members of Nat plus all the negative integers. The method *asString* is implemented as an overloaded method: if the receiver object is nonnegative, then the declaration in Nat is used because it is more specific, but if the receiver object is negative, then the declaration in Int is used because the one in Nat is not applicable. Both declarations are part of the intended algorithm for converting a value of type Int to a string.

Now consider the same example using top-level functions rather than dotted methods as in the right side of Figure 1. When another component M imports the component, every function declaration and every method declaration is, in effect, required to defer (that is, possibly dispatch) to other functions or methods that are accessible within component M, applicable to the arguments received, and more specific than the function or method declaration.

Finally, the example in Figure 2 uses functional methods rather than dotted methods or top-level functions. 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

```
component IntegerToStringFunctionalMethod
  trait Int
    asString(self) = "-" ‖ asString(− self)
  end
  trait Nat extends Int
    asString(self) =
      if self < 10 then "0123456789"[self : self]
      else
        q = self ÷ 10
        r = self − 10 q
        asString(q) ‖ asString(r)
      end
  end
end IntegerToStringFunctionalMethod
```

Fig. 2. Overloaded functional methods

```
component MyProgram
  import IntegerToStringFunctionalMethod.{ asString }
  asString(x : Boolean): String = if x then "true" else "false" end
  run() = println (asString (42))
end MyProgram
```

Fig. 3. Main component importing overloaded functional methods

in the same per-component namespace as top-level functions, and the dispatch model that works for top-level functions while preserving component modularity also works for functional methods. Thus, we often find ourselves preferring functional methods to other functionals in Fortress.

We might then have another component that can be run as a "main program", which imports only the *asString* functional method from IntegerToStringFunctionalMethod as shown in Figure 3. Note that the imported *asString* functional method is overloaded with the top-level *asString* function in the importing component. As the example shows, fine-grained imports of functionals affect which functional to invoke at run time.

2.4 Static Overloading Rules

To guarantee type safety in the presence of all the features described so far, we place static restrictions on overloaded functional declarations. We require every pair of overloaded functional declarations to satisfy the following two properties:

1. **The Subtype Rule**
   Whenever the parameter type of one is a subtype of the parameter type of the other, the return type of the first must also be a subtype of the return type of the second.
2. **The Meet Rule**
   Whenever the parameter types of the two declarations have a common lower bound (i.e., a common subtype), there is a unique declaration for the same functional whose parameter type is the greatest lower bound of the parameter types of the two declarations.

The flip side of the Meet Rule is this:

3. **The Exclusion Rule**
   Whenever the parameter types of two declarations are disjoint, the pair is a valid overloading.

Based on our experience with Fortress, these rules are not difficult to obey, especially because the compiler gives useful feedback.

While the Subtype Rule and the Meet Rule are necessary for type soundness, the Exclusion Rule enlarges a set of valid overloading. Fortress allows programmers to declare that two traits *exclude* each other (that is, no object can belong to both traits) and it also provides structural exclusion relationships. Each object type implicitly excludes every other object type because no object can extend them; an object type implicitly excludes types that are not its ancestors in the trait hierarchy. Also, a type of one parameter list excludes a type of another parameter list, if the sizes of the parameter lists are different, or any of their constituent types at the same position exclude each other. For simplicity, we consider only structural exclusion relationships rather than declared exclusion relationships in this paper. For details of a type system that handles declared exclusion relationships, see [3].

Checking the overloading rules consists of two parts: the overloaded methods in a trait are checked for validity, and overloaded top-level functions and functional methods in a component are checked for validity. In earlier work, we proposed an informal description of such overloading rules only for top-level functions and functional methods where top-level functions and functional methods may not be overloaded [2], and we proved that the overloading rules guarantee no ambiguous calls at run time [11]. In this paper, we present a system which allows overloading between top-level functions and functional methods, and fine-grained namespace control via modules and selective imports in Section 3, and we mechanize the system and the proof of its type soundness property in Coq in Section 4.

3 Calculus: MFFMM

We now define MFFMM (Modular Featherweight Fortress with Multiple dispatch and Multiple inheritance), a core calculus for Fortress. Due to space limitations, we describe only its central parts in prose in this section. The full syntax and semantics of MFFMM are available in our companion report [12].

$$
\begin{array}{lcl}
p & ::= & \overline{comp}\ \overline{\texttt{import}\ M.\{\overline{i}\}}\ \overline{d}\ e \\
comp & ::= & \texttt{component}\ M\ \overline{\texttt{import}\ M.\{\overline{i}\}}\ \overline{d}\ \texttt{end} \\
i & ::= & C \mid m \\
d & ::= & td \mid od \mid fd \\
td & ::= & \texttt{trait}\ T\ \texttt{extends}\ \{\overline{T}\}\ \overline{md}\ \texttt{end} \\
od & ::= & \texttt{object}\ O(\overline{f\colon C})\ \texttt{extends}\ \{\overline{T}\}\ \overline{md}\ \texttt{end} \\
fd & ::= & m(\overline{x\colon C})\colon C = e \\
md & ::= & m(\overline{x\colon C}\ \texttt{self}^{?}\ \overline{x\colon C})\colon C = e \\
e & ::= & x \mid \texttt{self} \mid O^{M}(\overline{e}) \mid e.f \mid e.m(\overline{e}) \mid m^{M}(\overline{e})
\end{array}
$$

Fig. 4. Syntax of MFFMM

3.1 Syntax and Adjustments for Components

The syntax of MFFMM is shown in Figure 4. The metavariables M ranges over component names; T ranges over trait names; O ranges over object names; C ranges over trait and object names; m ranges over function and method names; f ranges over field names; and x ranges over method and function parameter names. We write $\overline{x}$ as shorthand for a possibly empty sequence $x_1, \cdots, x_n$.

A program p is a sequence of component declarations followed by a designated "main" component. A component declaration consists of its name M, a sequence of import statements, and a sequence of top-level declarations. The main component is different from the other components in that it does not specify a name and it has a top-level expression denoting the "run" function of the program. For simplicity, we assume that the name of the main component is Mc, and it must not be the name of any other component. An import statement may import a set of imported items; an imported item is either a type name C or a (possibly overloaded) function name m. A top-level declaration may be a trait declaration, an object declaration, or a function declaration.

A trait or object declaration may extend multiple supertraits; it inherits the methods provided by its extended traits. It may include method declarations; a method declaration is either a dotted method declaration or a functional method declaration depending on the absence or presence of `self` in its parameter list. An object declaration may include field declarations as its value parameters. Traits and objects are collectively called types. While dotted methods in a type may be overloaded with only other dotted methods in the same type, functional methods in a type may be overloaded with not only other functional methods in the same type but also other functional methods and top-level functions in the component textually enclosing the type. Note that a dotted method may not be overloaded with a functional method nor with a top-level function.

An expression is either a variable reference, an object construction, a field access, a method invocation, or a function call. A variable reference is either a parameter name x or `self`. Note that the object name in an object construction and the function name in a function call are annotated by a component name M. As we discuss below, evaluation of a program consists of evaluation of expressions in various components, and evaluating an expression requires the component name textually enclosing the expression.

```
component MatrixLibrary
  trait Matrix extends {Object} end
  object UnitMatrix() extends {Matrix} end
end
component MyMatrixLibrary
  import MatrixLibrary.{Matrix}
  object UnitMatrix() extends {Matrix} end
  object GenUnitMatrix() extends {Object}
    gen(): UnitMatrix = UnitMatrix()
  end
end
import MatrixLibrary.{Matrix, UnitMatrix}
import MyMatrixLibrary.{GenUnitMatrix}
asString(x: Matrix): String = "Matrix"
asString(x: UnitMatrix): String = "UnitMatrix"
asString(GenUnitMatrix().gen())
```

Fig. 5. MFFMM program before the annotation phase

To support the component system with selective imports, MFFMM uses: *i)* annotations of enclosing component names on function calls and object constructions and *ii)* $actualTy_p(M, C)$, a pair of the type C appearing in M and its defining component, rather than C to take into account the component M in which the type C is defined. When a program consists of multiple components, evaluation of the program may require evaluation of the expressions in other components than the main component. Consider the example in Figure 5 where the object UnitMatrix in the component MatrixLibrary and the object UnitMatrix in the component MyMatrixLibrary are distinct types. The main component includes $asString\big(\text{GenUnitMatrix}().gen()\big)$ which evaluates to $asString\big(\text{UnitMatrix}()\big)$. Note that UnitMatrix here is the object defined in MyMatrixLibrary rather than in MatrixLibrary while the imported type UnitMatrix is imported from MatrixLibrary rather than from MyMatrixLibrary.

In order to evaluate the function call correctly, we need two pieces of information. First, we need to know in which component the function call of $asString$ textually appears to collect a set of accessible (or visible) function declarations for the function call to decide which function to call. Secondly, we need to know in which component the object construction GenUnitMatrix() textually appears to know the argument type of the function call to decide which function to call. Such information is syntactically available and a simple preprocessing phase can annotate each function call and object construction with its textually enclosing component name. The annotated component names on function calls and object constructions denote the actual use sites of the functions and objects. For example, the preprocessing phase rewrites the gen method declaration in the object GenUnitMatrix as follows:

$$gen(): \text{UnitMatrix} = \text{UnitMatrix}^{\text{MyMatrixLibrary}}()$$

and the function call in the main component as follows:

$$asString^{\mathrm{Mc}}\big(\mathrm{GenUnitMatrix}^{\mathrm{Mc}}().gen()\big)$$

At run time, one step evaluation of the call would lead to the following:

$$asString^{\mathrm{Mc}}\big(\mathrm{UnitMatrix}^{\mathrm{MyMatrixLibrary}}()\big)$$

which allows to select the correct function declaration to call:

$$asString(x\colon \mathrm{Matrix})\colon \mathrm{String} = \text{“}\texttt{Matrix}\text{”}$$

and evaluates to “`Matrix`” as desired.

As the example shows, MFFMM allows programmers to define types with the same name if they are in different components. While they do not produce any name conflicts syntactically, they may lead to name conflicts during type checking; types that are not defined nor imported by a component may not be explicitly used by the programmer but they may be implicitly available during type checking. For example, even though $\mathrm{UnitMatrix}$ in $\mathrm{MyMatrixLibrary}$ is not explicitly imported by the main component, it is available for type checking $\mathrm{GenUnitMatrix}^{\mathrm{Mc}}().gen()$. Thus, type names are not enough for identifying types but a pair of a type name and its defining component name can serve as a true identity for a type.

3.2 Static Semantics and Overloading Rules

In this section, we describe only the key rules of the static semantics especially for checking valid overloading; the full semantics is available in our companion report [12]. Type checking a program consists of checking its constituent component declarations and checking import statements, top-level declarations, overloaded functions, and the top-level expression of the main component. Checking function declarations consists of three parts: checking every pair of distinct declarations between top-level functions, between top-level functions and functional methods, and between functional methods.

Two top-level function declarations are valid if they satisfy any of the overloading rules: if their parameter types are disjoint, if one parameter type is more specific than the other, or if there exists a tie-breaking declaration between them.

A top-level function declaration and a functional method declaration in the same component may be a valid overloading if their parameter types are disjoint or the functional method declaration is more specific than the top-level function declaration. In other words, there must not be any top-level function whose signature is more specific than that of an overloaded functional method. The reason for this additional restriction is that for functional methods (unlike for top-level functions), we cannot statically determine all the declarations that are dynamically applicable. (Indeed, this is the reason the overloading rules are defined as they are, rejecting some sets of overloaded declarations even though there is no static ambiguity.)

For example:

```
component MoreSpecificTopLevel
  trait Matrix extends {Object}
```

```
    multiply(self, z: Object) = "In Matrix"
  end
  object SparseMatrix() extends {Matrix}
    multiply(self, z: ℤ) = "In SparseMatrix"
  end
  multiply(m: Matrix, z: ℤ) = "At Top-Level"
  double(m: Matrix) = multiply (m, 2)
  run() = double (SparseMatrix())
end MoreSpecificTopLevel
```

the signatures of the functional declarations above are as follows:

```
multiply(m: Matrix, z: Object)      // functional method from Matrix
multiply(m: SparseMatrix, z: ℤ)     // functional method from SparseMatrix
multiply(m: Matrix, z: ℤ)           // top-level function
```

Because the static type of the first argument m of the function call *multiply* $(m, 2)$ is Matrix, the functional method declared in SparseMatrix is not statically applicable to the call. Among the other two applicable declarations, the top-level function is the most specific statically applicable one. However, at run time, because m is SparseMatrix, the functional method declared in SparseMatrix is also applicable to the call and it is even the most specific one. Therefore, if we allow top-level functions to be more specific than their overloaded functional methods, we need to consider not only the top-level functions but also the functional methods and more specific functional methods overriding them at run time, which burdens the performance of dynamic dispatch.

Instead, if we require functional method declarations be more specific than the overloaded top-level function declarations, the overloaded declarations in the above example are invalid. Let us consider the slightly revised example:

```
component LessSpecificTopLevel
  trait Matrix extends {Object}
    multiply(self, z: ℤ) = "In Matrix"
  end
  object SparseMatrix() extends {Matrix}
    multiply(self, z: ℤ) = "In SparseMatrix"
  end
  multiply(m: Matrix, z: Object) = "At Top-Level"
  double(m: Matrix) = multiply (m, 2)
  run() = double (SparseMatrix())
end LessSpecificTopLevel
```

Now the signatures of the overloaded declarations are as follows:

```
multiply(m: Matrix, z: ℤ)           // functional method from Matrix
multiply(m: SparseMatrix, z: ℤ)     // functional method from SparseMatrix
multiply(m: Matrix, z: Object)      // top-level function
```

Among the applicable declarations to the call $multiply\,(m, 2)$, the functional method declared in Matrix is the most specific statically applicable one. Because we restrict top-level functions from being more specific than any overloaded functional method, for any call to which both a top-level function and a functional method are applicable (statically or dynamically), the top-level function will never be the most specific dynamically applicable declaration: there will always be some dynamically applicable functional method declaration that is more specific (because of the Meet Rule). Thus, at run time, we need to consider only the functional method and the more specific functional methods overriding it without considering any top-level functions, which largely reduces the set of the candidate methods to investigate.

Finally, two functional methods in a component which may be defined in different types are valid if the parameter types of them are disjoint or they have `self` in the same position of their parameter lists. When types A and B provide a functional method with the same name and `self` in the same position, and another type C extends both A and B inheriting both functional methods, then C itself should provide a disambiguating definition, and this is checked by the overloading rules when compiling C, because at that point, the declarations in both A and B are visible (since C extends both A and B).

To see why we need this requirement for the `self` position, consider the following Matrix and Vector example:

```
trait Matrix extends {Object}
  multiply(self, y: Vector) = "In Matrix"
end
trait Vector extends {Object}
  multiply(x: Matrix, self) = "In Vector"
end
```

Since both functional method declarations have the same signature, any call to *multiply* with two arguments of types Matrix and Vector is ambiguous. For example:

```
object MatrixVector extends {Matrix, Vector} end
```

because MatrixVector is both Matrix and Vector, the following call is ambiguous:

$$multiply(\text{MatrixVector}, \text{MatrixVector})$$

because both functional method declarations from Matrix and Vector are accessible, applicable, and equally specific. With the restriction of the same position for `self`, all the functional method declarations in a valid overloading set whose parameter types are not disjoint have the `self` parameter in the same position and the type of `self` is its enclosing trait. Therefore, we can guarantee that a functional method declaration chosen with more specific argument types is defined in a subtype of the owner of a functional method declaration chosen with less specific argument types.

3.3 Dynamic Semantics and Functional Dispatch

Evaluation of a method invocation $O^M(\overline{v}).m(\overline{v'})$ is conventional except that we first find the component M' that defines O by $actualTy_p(M, O)$, which effectively computes visibility from M. Among a set of visible dotted methods in O defined in M',

we collect a set of applicable methods to the call using the dynamic types of the arguments, and select the most specific one from the set.

Due to the unique characteristic of functional methods, evaluation of a function call $m^M(\overline{v})$ requires an additional step to find the most specific functional declaration. Among a set of visible functionals in M, we first collect a set of applicable functionals to the call using the dynamic types of the arguments, just like for the method invocation case. Then, if the candidate set consists of only top-level functions, we simply select the most specific one from the set. However, if the candidate set includes any functional method declaration, we collect yet another set of visible functional method declarations from the enclosing trait of the functional method declaration. We then perform the second dispatch by collecting a set of applicable functional method declarations from the set and selecting the most specific one from the set.

Let us revisit the *multiply* example in Section 3.2 one more time. For the function call $multiply\,(m, 2)$, the static types of the arguments are $(\mathrm{Matrix}, \mathbb{Z})$ and the dynamic types of the arguments are $(\mathrm{SparseMatrix}, \mathbb{Z})$. While the signatures of the statically applicable declarations are as follows:

$multiply(m\colon \mathrm{Matrix}, z\colon \mathbb{Z})$ // functional method from Matrix
$multiply(m\colon \mathrm{Matrix}, z\colon \mathrm{Object})$ // top-level function

the signatures of the dynamically applicable declarations include the following as well:

$multiply(m\colon \mathrm{SparseMatrix}, z\colon \mathbb{Z})$ // functional method from $\mathrm{SparseMatrix}$

Because the dynamically applicable declarations include functional method declarations, we collect a set of visible functional method declarations from Matrix and $\mathrm{SparseMatrix}$ so that we do not miss any functional method declarations that are not visible at compile time but are visible at run time. Because we collect visible functional method declarations from static types at compile time and from dynamic types at run time, and because run-time types are more specific than compile-time types, more functional method declarations may be visible at run time. Then, we perform the second dispatch by collecting a set of applicable functional method declarations from the set and selecting the most specific one from the set.

This dispatch mechanism always selects the unique most specific function to each call. First, if the applicable functional set for a function call has no functional methods, all the candidates in the set are top-level functions, and the validity of top-level function overloading guarantees that there always exists the most specific function for the call. Secondly, if the applicable functional set for a function call has more than one functional methods, the functional methods are more specific than any of the top-level functions in the set and the most specific one from the set is a functional method. Note that because the static semantics guarantees that the overloaded top-level functions are less specific than the overloaded functional methods, we do not need to consider top-level functions here, which makes this second dispatch more efficient. Finally, because all the functional methods in the visible functional set have `self` in the same position in their parameter declarations unless their parameter types are disjoint, and the type of `self` is the owner type of each functional method, the dynamic type of the argument corresponding to `self` provides all the applicable functional methods to the call

including the ones in a supertype of the dynamic type. Given all these conditions, the dispatch mechanism always selects the unique dynamically most specific function to a call.

4 Properties and Coq Mechanization

We fully mechanized MFFMM and its type safety proof in Coq. Our mechanization is based on the `metatheory` library developed by De Fraine *et al.* [9]. The Coq mechanization is very close to MFFMM so that one can easily find the corresponding declarations and rules between them. The few differences between them are mostly Coq-specific implementation details, which we omit in this paper.

We proved two traditional theorems for type safety of MFFMM:

Theorem 1 (Progress). *Suppose that p is well typed. If an expression e in p has type (M, C), then e is a value or there exists some e' such that e evaluates to e'.*

Theorem 2 (Preservation). *Suppose that p is well typed. If an expression e in p has type (M, C) and e evaluates to e', then e' has type (M', C') where (M', C') is a subtype of (M, C).*

To prove the type safety of MFFMM, we also proved that every functional call in a well-typed program is uniquely dispatched. Due to space limitations, we refer the interested readers to our companion report [12].

While Fortress provides separate compilation by components and *APIs* where interfaces between components are described by APIs, we omit APIs for simplicity in this paper. Note that our formalization captures separate compilation even without APIs because components must import declarations from other components to make them visible, and validity judgments are applied only to visible declarations. This is unlike Java, for example, in which importing only enables the use of unqualified names. Thus, each component can be checked separately with references only to those declarations from components that it explicitly imports.

As we discussed in Section 3.1, the key aspects of MFFMM to support components, which are essential in proving the type safety, are to annotate textually enclosing component names to function calls and object constructions and to use $actualTy_p(M, C)$ to denote a type C defined in a component M. Because evaluating an expression in a component may require evaluation of other expressions in different components, evaluation rules may keep track of the component where the current evaluation occurs and the component that the control goes back to when the current evaluation normally finishes. Because small-step operational semantics such as our dynamic semantics are not well suited for keeping track of such surrounding information, we use the annotations of textually enclosing component names to represent such information. Also, $actualTy_p(M, C)$ explicitly denotes the true identities of types that are necessary to distinguish between types of the same name from different components.

The Coq mechanization of MFFMM is based on our previous work [11] on FFMM, but we did not reuse much of the FFMM Coq code mainly because the design of FFMM lacks extensibility. We organized the syntax and semantics of MFFMM in a

modular way so that the new features such as functional methods and components are represented seamlessly and possible future changes in the calculus can be integrated smoothly. The Coq mechanization is approximately 9,000 lines and it is available online [10].

5 Related Work

Millstein and Chambers introduce the *Dubious* language [14], which provides symmetric dynamic multimethod dispatch while allowing a program to be divided into modules that can be separately type-checked statically. Our 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 our work supports dynamically created objects and state; Dubious does not provide disjoint relations between types and it requires every multimethod to have a principal type[1], thus it cannot support multimethods that take different numbers of arguments or otherwise do not have a principal type, nor can it allow the position of the owner to vary, whereas Fortress enlarges a set of valid overloadings thanks to disjoint type relations; 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 import statements allow fine-grained selective import of any parts of a component—in particular, it is possible to import only selected functional methods of a trait, rather than all methods. Other languages from the same research group that are similarly closely related to the present work are Cecil [5], EML [13], MultiJava [7], and Relaxed MultiJava [15]. A block-structured variant of Cecil, BeCecil [6], supports multimethod declarations in a nested scope, limiting their visibility to the scope. However, BeCecil does not support a module system, thus it does not support modular type checking.

Odersky, Wadler and Wehr's System O [17] supports overloaded declarations with completely different type signatures, and is modular in the sense that it has the Hindley/Milner type system. The system ensures no ambiguities by putting a simple restriction on type classes, but it requires that overloaded functions should be dispatched on the first parameter while Fortress allows multiple dispatch.

Scala provides a way to omit some arguments at a method call if they are bound to *implicit* parameters [18]. Selecting the most specific implicit parameter that applies in the method call is similar to overloading resolution in Fortress. While Fortress prohibits any possibilities of ambiguous calls at functional declaration sites, Scala statically rejects ambiguous calls at method use sites.

In Haskell typeclasses [23], overloaded functions must be contained in some type class, and their signatures must vary in exactly the same structural position. Typeclasses are ill-suited for functions lacking uniform variance in the domain and range, for example. Such behavior is consistent with the static, type-based dispatch of Haskell, but it would lead to irreconcilable ambiguity in the dynamic, value-based dispatch of Fortress. While Fortress supports fine-grained imports of overloaded declarations, all instance

[1] The parameter and return types of any declaration for a multimethod must be subtypes of their corresponding types in the principal type.

declarations in Haskell are globally visible, and each declaration should check that it does not overlap with any of the others.

Type safety proofs for several programming languages are mechanized in various proof assistant tools. Our previous work [11] mechanizes type safety proofs of core calculi for Fortress in Coq, and the present work is an extension of them especially with a component system, top-level functions, functional methods, and overloading between top-level functions and functional methods. Strniša *et al.* [20] introduce a formal calculus for Java with a module system, and mechanize its type safety proof using Isabelle/HOL [16]. The calculus does not provide overloading, and references across module boundaries use fully qualified names, which amounts to requiring programmers to use actual types. None of the calculi supports a module system, and the technique requires a calculus to have placeholders for future extension.

6 Conclusion

Namespace control in object-oriented languages is tricky: On one hand, we want to inherit method declarations implicitly and be able to override and overload them. On the other hand, we want to control access to specific methods by controlling where their names are in scope. Functional methods provide an effective solution: they are inherited like conventional dotted methods, but their visibility is controlled by components with selective imports that allow fine-grained namespace control, like top-level functions, with which they can be overloaded.

Functional methods are an effective approach to solving the operator method problem. The advantage over dotted methods is that any argument position may serve as the receiver; the advantage over ordinary functions is that a trait may declare a set of overloaded operators with disjoint parameter types. Ensuring the existence of the unique most specific functional declaration for a call in the presence of overloading between three kinds of functional declarations with symmetric multiple dispatch is tricky. A component system with selective imports introduces yet another problem that we should consider the *hidden* functional methods in traits when we select the most specific top-level function or functional method for a function call. We have shown how these features work in a manner that interacts well with namespace control. To guarantee that such features do not cause any undefined or ambiguous calls at run time, we present a core calculus for Fortress and fully mechanize its type safety proof in Coq.

In the future, we plan to extend this framework to parametrically polymorphic types, and more type relations such as explicit type exclusion and comprises relations [3].

Acknowledgments. This work is supported in part by Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grants NRF-2011-0016139 and NRF-2008-0062609).

References

1. Allen, E., Chase, D., Hallett, J.J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr., G.L., Tobin-Hochstadt, S.: The Fortress Language Specification Version 1.0 (March 2008)

2. Allen, E., Hallett, J.J., Luchangco, V., Ryu, S., Steele, G.: Modular multiple dispatch with multiple inheritance. In: Proc. ACM Symposium on Applied Computing (2007)
3. Allen, E., Hilburn, J., Kilpatrick, S., Luchangco, V., Ryu, S., Chase, D., Steele Jr., G.L.: Type-checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance. In: OOPSLA (2011)
4. Castagna, G., Ghelli, G., Longo, G.: A calculus for overloaded functions with subtyping. In: LFP (1992)
5. Chambers, C.: Object-oriented multi-methods in Cecil. In: Madsen, O.L. (ed.) ECOOP 1992. LNCS, vol. 615, pp. 33–56. Springer, Heidelberg (1992)
6. Chambers, C., Leavens, G.T.: Bececil, a core object-oriented language with block structure and multimethods: Semantics and typing. In: Proceedings of the 4th International Workshop on Foundations of Object Oriented Languages (1997)
7. Clifton, C., Millstein, T., Leavens, G.T., Chambers, C.: MultiJava: Design rationale, compiler implementation, and applications. ACM TOPLAS 28(3), 517–575 (2006)
8. The COQ Development Team. The COQ Proof Assistant, `http://coq.inria.fr/`
9. De Fraine, B., Ernst, E., Südholt, M.: Cast-Free Featherweight Java (2008), `http://soft.vub.ac.be/~bdefrain/featherj`
10. Kim, J.: MFFMM in COQ (2012), `http://plrg.kaist.ac.kr/_media/research/software/mffmm_in_coq.tar.gz`
11. Kim, J., Ryu, S.: COQ mechanization of Featherweight Fortress with multiple dispatch and multiple inheritance. In: Proceedings of the First International Conference on Certified Programs and Proofs (2011)
12. Kim, J., Ryu, S.: MFFMM : Modular Featherweight Fortress with Multiple Dispatch and Multiple Inheritance (June 2013), `http://plrg.kaist.ac.kr/_media/research/publications/mffmm_calculus.pdf`
13. Millstein, T., Bleckner, C., Chambers, C.: Modular typechecking for hierarchically extensible datatypes and functions. ACM TOPLAS 26(5), 836–889 (2004)
14. Millstein, T., Chambers, C.: Modular statically typed multimethods. Information and Computation 175(1), 76–118 (2002)
15. Millstein, T.D., Reay, M., Chambers, C.: Relaxed MultiJava: balancing extensibility and modular typechecking. In: OOPSLA (2003)
16. Nipkow, T., Paulson, L.C., Wenzel, M.T.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
17. Odersky, M., Wadler, P., Wehr, M.: A second look at overloading. In: FPCA, pp. 135–146 (1995)
18. Oliveira, B.C.D.S., Moors, A., Odersky, M.: Type classes as objects and implicits. In: OOPSLA (2010)
19. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable Units of Behaviour. In: Cardelli, L. (ed.) ECOOP 2003. LNCS, vol. 2743, pp. 248–274. Springer, Heidelberg (2003)
20. Strniša, R., Sewell, P., Parkinson, M.: The Java module system: Core design and semantic definition. In: OOPSLA (2007)
21. Stroustrup, B.: The C++ Programming Language. Addison-Wesley, Reading (1986)
22. Szypersky, C., Omohundro, S., Murer, S.: Engineering a programming language: The type and class system of Sather. In: Gutknecht, J. (ed.) Programming Languages and System Architectures. LNCS, vol. 782, pp. 208–227. Springer, Heidelberg (1994)
23. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: POPL (1989)