Adding Functions and Modules to FFMM

Jieung Kim
with Sukyoung Ryu

Programming Language Research Group
Korea Advanced Institute of Science and Technology

Jan. 17, 2012

FFMM

- Featherweight Fortress with Multiple Dispatch and Multiple Inheritance
- A subset of the Fortress programming language
- Multiple dispatch
  To select the best method to call, the runtime engine uses the dynamic types of all the arguments
- Multiple inheritance
  A type may have multiple super types
- Two different kinds of type
  Traits: Types without field
  Objects: Leaves of type hierarchy containing field

FFMM Mechanization

Coq Mechanization of Featherweight Fortress with Multiple Dispatch and Multiple Inheritance

Jieung Kim and Sukyoung Ryu

Computer Science Department, KAIST
{gbali,sryu.cs}@kaist.ac.kr

Abstract. In object-oriented languages, overloaded methods with multiple dispatch extend the functionality of existing classes, and multiple inheritance allows a class to reuse code in multiple classes. However, both multiple dispatch and multiple inheritance introduce the possibility of ambiguous method calls that cannot be resolved at run time. To guarantee no ambiguous calls at run time, the overloaded method declarations should be checked statically.

In this paper, we present a core calculus for the Fortress programming language, which provides both multiple dispatch and multiple inheritance. While previous work proposed a set of static rules to guarantee no ambiguous calls at run time, the rules were parametric to the underlying programming language. To implement such rules for a particular language, the rules should be instantiated for the language. Therefore, to concretely realize the overloading rules for Fortress, we formally define a core calculus for Fortress and mechanize the calculus and its type safety proof in Coq.

Keywords: Coq, Fortress, overloading, multiple dispatch, multiple inheritance, type system, proof mechanization.

1 Introduction

Most object-oriented programming languages support method *overloading*: a method may have multiple declarations with different parameter types. Multiple method declarations with the same name can make the program logic clear and simple. When several of the overloaded methods are applicable to a particular call, the most specific applicable declaration is selected by the *dispatch mechanism*.

Several dispatch mechanisms exist for various object-oriented languages. For example, the Java^{TM} programming language [11] uses a single-dispatch mechanism, where the dynamic type of only a single argument (the receiver of the method) and the static types of the other arguments are considered for method selection. CLOS [9] uses asymmetric multiple dispatch, where the dynamic type of each argument is considered in a specified order (usually left to right) for method selection. Fortress [3] uses *symmetric multiple dispatch*, where the dynamic types of all the arguments are equally considered. Because previous work [24,26,4,13] observed that using static types of arguments or a particular order of method parameters for method selection often produces confusing results, we focus on symmetric multiple dispatch throughout this paper.

Multiple inheritance lets a type have multiple super types, which allows the type to reuse code in its multiple super types, and permits more type hierarchies than what

J.-P. Jouannaud and Z. Shao (Eds.): CPP 2011, LNCS 7086, pp. 264–279, 2011.
© Springer-Verlag Berlin Heidelberg 2011

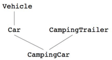

Fig. 1. Example type hierarchy

are allowed in single inheritance. While multiple inheritance provides high expressive power, it has well-known problems such as "name conflicts" and "state conflicts" [28]. Several object-oriented languages support multiple inheritance by addressing these problems in different ways. For example, C++ [29] requires programmers specify how to resolve conflicts between inherited fields. Scala [27] supports multiple inheritance via *traits* [28], where the order of super traits resolves any conflicts. Fortress [3] also provides multiple inheritance via traits, but the order of super traits does not affect the language semantics. Instead, Fortress traits do not include any fields, which removes the possibility of state conflicts. Similarly to the dispatch mechanism, we focus on symmetric multiple inheritance in this paper.

However, both multiple dispatch and multiple inheritance introduce the possibility of ambiguous method calls that cannot be resolved at run time. Consider a type hierarchy illustrated in Figure 1 in a language with multiple dispatch and multiple inheritance. The following overloaded method declarations:

```
collide(Car c, CampingCar cc)
collide(CampingCar cc, Car c)
```

introduce a possibility of an ambiguous method call due to multiple dispatch. For a method call `collide(cc1, cc2)` where both `cc1` and `cc2` have the `CampingCar` type at run time, we cannot select the best method to call because none of the `collide` method declarations is more specific than the other. Likewise, the following overloaded method declarations:

```
lightOn(Car c)
lightOn(CampingTrailer ct)
```

introduce a possibility of an ambiguous method call due to multiple inheritance. For a method call `lightOn(cc)` where `cc` has the `CampingCar` type at run time, we cannot select the best method to call because none of the `lightOn` method declarations is more specific than the other.

To break ties between ambiguous method declarations to a call, there should exist a disambiguating method declaration that is more specific than the ambiguous declarations and also applicable to the call. For example, if we add the following declaration to the above set of `collide` method declarations:

```
collide(CampingCar cc1, CampingCar cc2)
```

What Is Next?

Adding functions and modules
to FFMM

Functions and Modules

Adding Functions

- Enhances function extensibility
- Allows programmers to express mathematical notation as closely as possible

Adding Modules

- Provides a unit of compilation and a unit of code distribution
- Allows programmers to develop large software mostly independently
- Allows programmers to handle namespace efficiently

Functional Declarations in Fortress

- Dotted methods
- Top-level functions
- Functional methods

Dotted Methods

- Similar to method declarations in JAVA™

```
trait Vector
    multiply(m: Matrix): Vector = ...
end

trait Matrix excludes Vector
    multiply(v: Vector): Vector = ...
end
```

$$m.multiply(v)$$
$$v.multiply(m)$$

Top-level Functions

- Allow to extend an existing class with entirely new functionality

```
trait Vector end

trait Matrix excludes Vector end

multiply(m: Matrix, v: Vector): Vector = ...
multiply(v: Vector, m: Matrix): Vector = ...
```

$$multiply(m,\ v)$$
$$multiply(v,\ m)$$

Functional Methods

- Allow one `self` parameter in a parameter list
- Inherited to subtypes
- Overloaded with top-level functions

```
trait Vector

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

$$multiply(m,\ v)$$
$$multiply(v,\ m)$$

Functional Methods

- Allow one `self` parameter in a parameter list
- Inherited to subtypes
- Overloaded with top-level functions

```
trait Vector

trait Matrix excludes Vector
    opr ·(self, v: Vector): Vector = ...
    opr ·(v: Vector, self): Vector = ...
end
```

$$m \cdot v$$
$$v \cdot m$$

Module System of Fortress

- Fortress divides a program into *components*
- Components may import declarations in other components via APIs, which serve as "interfaces" of the components.
- Each component is modularly checked at compile time

Module System

```
component A

end
```

- Declarations in component

Module System

```
component A
    trait Matrix end




end
```

- Declarations in component
  - Trait declarations

Module System

```
component A
    trait Matrix end

    object SparseMatrix extends Matrix
        getSize(): ℕ = ...
        multiply(self, m: Matrix) = ...
    end

end
```

- Declarations in component
  - Trait declarations
  - Object declarations

Module System

```
component A
    trait Matrix end

    object SparseMatrix extends Matrix
        getSize(): ℕ = ...
        multiply(self, m: Matrix) = ...
    end

    multiply(m: Matrix, m: Matrix)
end
```

- Declarations in component
  - Trait declarations
  - Object declarations
  - Top-level functions

Module System

```
component A
    trait Matrix end

    object SparseMatrix extends Matrix
        getSize(): ℕ = ...
        multiply(self, m: Matrix) = ...
    end

    multiply(m: Matrix, m: Matrix)
end

component B
    import A.{                    }
    ...
end
```

- Declarations in component
  - Trait declarations
  - Object declarations
  - Top-level functions
- Import items

Module System

```
component A
    trait Matrix end

    object SparseMatrix extends Matrix
        getSize(): ℕ = ...
        multiply(self, m: Matrix) = ...
    end

    multiply(m: Matrix, m: Matrix)
end

component B
    import A.{SparseMatrix            }
    ...
end
```

- Declarations in component
  - Trait declarations
  - Object declarations
  - Top-level functions
- Import items
  - Types

Module System

```
component A
    trait Matrix end

    object SparseMatrix extends Matrix
        getSize(): ℕ = ...
        multiply(self, m: Matrix) = ...
    end

    multiply(m: Matrix, m: Matrix)
end

component B
    import A.{SparseMatrix, multiply}
    ...
end
```

- Declarations in component
  - Trait declarations
  - Object declarations
  - Top-level functions
- Import items
  - Types
  - Functions

Restrictions Related to Functional Declarations

- A top-level function declaration must not be more specific than a functional method declaration with the same name.

Restrictions Related to Functional Declarations

- A top-level function declaration must not be more specific than a functional method declaration with the same name.

```
trait Matrix
    multiply(self, z: ℤ) ...
end

multiply(m: Matrix, t: Object) ...

object SparseMatrix extends Matrix
    multiply(self, n: ℕ) ...
end

multiply(sm: SparseMatrix, z: ℤ) ...
```

multiply (*m* , *3*)

Restrictions Related to Functional Declarations

- A top-level function declaration must not be more specific than a functional method declaration with the same name.

```
trait Matrix
    multiply(self, z: ℤ) ...
end

multiply(m: Matrix, t: Object) ...

object SparseMatrix extends Matrix
    multiply(self, n: ℕ) ...
end

multiply(sm: SparseMatrix, z: ℤ) ...
```

$$multiply(m,\ 3)$$
$$\Downarrow$$
$$multiply(sm,\ 3)$$

Restrictions Related to Functional Declarations

- A top-level function declaration must not be more specific than a functional method declaration with the same name.

```
trait Matrix
    multiply(self, z: ℤ) ...
end

multiply(m: Matrix, t: Object) ...

object SparseMatrix extends Matrix
    multiply(self, z: ℤ) ...
    multiply(self, n: ℕ) ...
end
```

multiply (*m* , *3*)

Restrictions Related to Functional Declarations

- A top-level function declaration must not be more specific than a functional method declaration with the same name.

```
trait Matrix
    multiply(self, z: ℤ) ...
end

multiply(m: Matrix, t: Object) ...

object SparseMatrix extends Matrix
    multiply(self, z: ℤ) ...
    multiply(self, n: ℕ) ...
end
```

$$multiply(m,\ 3)$$
$$\Downarrow$$
$$multiply(sm,\ 3)$$

Restrictions Related to Functional Declarations

- `self` parameters must be in the same position of two functional method declarations if they do not satisfy the exclusion rule or the subtype rule.
- Inherited functional methods must be checked for overloading rules even when they are not explicitly imported.

Calculus

m	function and method name	f	field name
T	trait name	O	object name

x function/method parameter name

p	::=	$\overrightarrow{comp}$ `import` $\overrightarrow{M.\{\overrightarrow{i}\}}$ e	progra
$comp$	::=	`component` M $\overrightarrow{\texttt{import}\ M.\{\overrightarrow{i}\}}$ $\overrightarrow{d}$ `end`	compo
i	::=	τ	import
	$\mid$	m	
d	::=	td	top-lev
	$\mid$	od	
	$\mid$	fd	
τ	::=	T	type
	$\mid$	O	
e	::=	x	express
	$\mid$	`self`	
	$\mid$	$O(\overrightarrow{e})$	
	$\mid$	$e.f$	
	$\mid$	$e.m(\overrightarrow{e})$	
	$\mid$	$m(\overrightarrow{e})$	
fd	::=	$m(\overrightarrow{x\colon\tau})\colon\tau = e$	top-lev
md	::=	$m(\overrightarrow{x\colon\tau}$ `self`$^?$ $\overrightarrow{x\colon\tau})\colon\tau = e$	methoc
td	::=	`trait` T `extends` $\{\overrightarrow{T}\}$ $\overrightarrow{md}$ `end`	trait de
od	::=	`object` $O(\overrightarrow{f\colon\tau})$ `extends` $\{\overrightarrow{T}\}$ $\overrightarrow{md}$ `end`	object

Figure 1: Syntax

Valid top-level function declarations: $\boxed{p \vdash validTopFunc(M)\ /\ p \vdash validTopFunc'(M)\ /\ p \vdash validTopFunc''(M)}$

$$[\textsc{ValidTopFunc}]\quad \frac{p \vdash validTopFunc'(M) \qquad p \vdash validTopFunc''(M)}{p \vdash validTopFunc(M)}$$

$$[\textsc{ValidTopFunc}']\quad \frac{\begin{array}{l}\forall\{(M,\text{Object},fd),(M',\text{Object},fd')\} \subseteq visibleTopFunc_p(M^o).\\ \quad fd \neq fd',\ \text{(not same declaration)}\\ \quad fd = m(\overrightarrow{\_ : \tau^a}) : \tau^r\ \_,\\ \quad fd' = m(\overrightarrow{\_ : \tau^{a'}}) : \tau^{r'}\ \_,\\ \quad p \vdash valid(m, M, \text{Object}, \overrightarrow{\tau^a} \to \tau^r, M', \text{Object}, \overrightarrow{\tau^{a'}} \to \tau^{r'}, ProvidedFunc_p(M^o))\end{array}}{p \vdash validTopFunc'(M^o)}$$

$$[\textsc{ValidTopFunc}'']\quad \frac{\begin{array}{l}\forall(M,\text{Object},fd) \subseteq visibleTopFunc_p(M^o).\\ \quad \forall(M',\text{Object},md') \subseteq ProvidedFuncMeth'_p(M^o).\\ \qquad fd = m(\overrightarrow{\_ : \tau^a}) : \tau^r\ \_,\\ \qquad md' = m(\overrightarrow{\_ : \tau^{a'}}) : \tau^{r'}\ \_,\\ \qquad (|\overrightarrow{\tau^a}| \neq |\overrightarrow{\tau^{a'}}|) \vee (\overrightarrow{(M,\tau^a)} \neq \overrightarrow{(M',\tau^{a'})} \wedge p \vdash \overrightarrow{(M',\tau^{a'})} <: \overrightarrow{(M,\tau^a)})\end{array}}{p \vdash validTopFunc''(M^o)}$$

Figure 12: Rules to check valid top-level function sets

Conclusion

- Define a core calculus of the Fortress programming language
  - Three kinds of functional declarations
  - Multiple dispatch
  - Multiple inheritance
  - Modular checks
- Will mechanize its type safety using Coq

Any Questions?