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™ 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.

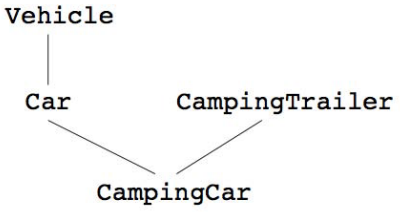

Fig. 1. Example type hierarchy

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)
```

the set would be a valid overloading, and the set of `lightOn` method declarations would be a valid overloading if it includes the following declaration:

```
lightOn(CampingCar cc)
```

Finding such a disambiguating method declaration is not always trivial. While the following set of method declarations seems to be valid because the third declaration is more specific than the first and second:

```
tow(Vehicle v, Car c)
tow(Car c, Vehicle v)
tow(CampingCar cc1, CampingCar cc2)
```

it is not a valid overloading: For a method call `tow(c1, c2)` where both `c1` and `c2` have the `Car` type at run time, we cannot select the best method to call because none of the first and the second declarations is more specific than the other, and the third declaration is not even applicable to `tow(c1, c2)`.

Previous work proposed a set of rules to check overloaded method declarations statically to guarantee no ambiguous calls at run time. The Fortress team designed such overloading rules and proved that the rules ensure that there exist no ambiguous calls at run time [4]. While the overloading rules were designed in the context of Fortress, they were not closely tied to a particular programming language. To make the rules more concrete enough to be clearly implementable, the team defined a calculus, *Core Fortress with Overloading* (CFWO) [3, Appendix A.2], but the calculus was not proved type sound.

In this paper, we present a core calculus for the Fortress programming language, *Featherweight Fortress with Multiple Dispatch and Multiple Inheritance* (FFMM), which provides both multiple dispatch and multiple inheritance. Unlike CFWO, FFMM does not support generic types which are orthogonal to the overloading rules as we discuss in Section 2. We formally define FFMM and mechanize it and its type safety proof in COQ. While proving the type safety of FFMM in COQ, we found a bug in CFWO and proposed a fix to it. Our COQ implementation is available online [20].

The remainder of this paper is organized thus. In Section 2, we discuss the overloading rules that are statically checked to guarantee no ambiguous calls at run time. We formally define such rules in the context of FFMM in Section 3, describe our mechanization of the calculus using COQ in Section 4, and present its type safety proof using COQ in Section 5. We share our experience of using COQ in the development of FFMM in Section 6 and discuss the related work in Section 7. Section 8 discusses future work of our research and concludes.

2 Overloading Rules

To make sure that one can always pick the best method to call among overloaded methods at run time, the Fortress team devised a set of overloading rules to check statically [4]. The rules determine whether a set of overloaded declarations is valid by

considering each pair of declarations in the set independently. A pair of declarations is a valid overloading if it satisfies one of the following rules:

1. **The Exclusion Rule**
   If the parameter types of the declarations are disjoint types, then the pair is a valid overloading.
2. **The Subtype Rule**
   If the parameter type of one declaration is a strict subtype of the parameter type of the other declaration, and the return type of the former is a subtype of the return type of the latter, then the pair is a valid overloading.
3. **The Meet Rule**
   If the parameter types of the declarations are not in the subtype relation, then the pair is a valid overloading if there is a declaration whose parameter type is an intersection type of the parameter types of the declarations.

We refer interested readers to the work of the Fortress team for detailed explanation of the rules [4]. They proved that the static rules placed on overloaded declarations are sufficient to guarantee no undefined nor ambiguous calls at run time. While the overloading rules are clearly described and the overloading resolution is proved to be unambiguous at run time, there still exists a gap between the rules and a particular programming language, Fortress.

Because the rules are specified independently for the underlying language, the Fortress team designed a calculus, CFwO [3, Appendix A.2], to describe the overloading rules more closely to the Fortress programming language. However, they did not prove the type safety of the calculus, and we found a bug in CFwO while we were working on our calculus as we discuss in Section 3.

To solve the problems arose from the previous approaches, we define a core calculus for the Fortress programming language, FFMM, which provides both multiple dispatch and multiple inheritance. Because CFwO is a strict extension of the *Basic Core Fortress* calculus [3, Appendix A.1], it includes generic types and top-level functions. While generic types can permit various interesting overloadings as the team presents in their recent work [5], the previous approaches assume that overloaded declarations should have type parameters that are identical (up to α-equivalence). Therefore, generic types and top-level functions are orthogonal to the overloading rules presented in [4] and FFMM does not support generic types nor top-level functions for simplicity. We formally define FFMM in Section 3, and we mechanize the definition and its type safety proof in Coq as we present in Sections 4 and 5.

3 FFMM: Featherweight Fortress with Multiple Dispatch and Multiple Inheritance

In this section, we formally define our calculus FFMM that we mechanize using Coq in the next section. Due to the space limitation, we describe only the rules that are closely related to overloaded methods in this paper. The full syntax, static semantics, and dynamic semantics of FFMM are available from a companion report [21].

p	::=	$\overrightarrow{d}\ e$	program
d	::=	`trait` T `extends` $\{\overrightarrow{T}\}$ $\overrightarrow{md}$ `end`	trait declaration
	|	`object` $O(\overrightarrow{f\colon \tau})$ `extends` $\{\overrightarrow{T}\}$ $\overrightarrow{md}$ `end`	object declaration
md	::=	$m(\overrightarrow{x\colon \tau})\colon \tau = e$	method declaration
e	::=	x	parameter
	|	`self`	self
	|	$O(\overrightarrow{e})$	object creation
	|	$e\,.\,f$	field access
	|	$e\,.\,m(\overrightarrow{e})$	method invocation
τ	::=	T	trait type
	|	O	object type

Fig. 2. Syntax of FFMM

3.1 Syntax

The syntax of FFMM is provided in Figure 2. The metavariables T ranges over trait names; O ranges over object names; m ranges over method names; f ranges over field names; and x ranges over method parameter names. We write $\overrightarrow{x}$ for a (possibly empty) sequence $x_1, \cdots, x_n$.

A program consists of a sequence of trait and object declarations followed by a single top-level expression. Following the precedent set by prior core calculi such as Featherweight Java (FJ) [19], we have abided by the restriction that all valid FFMM programs are valid Fortress programs except that certain simple syntactic abbreviations such as commas and semicolons must be expanded.

Trait and object declarations in a program may include method declarations. Object declarations may include field declarations, which are shown as value parameters. Both traits and objects may extend multiple traits; they inherit the methods provided by the extended traits. Method declarations in a trait or an object may have the same name; method declarations may be *overloaded*.

Valid expressions are parameter references, references to the special identifier `self` which is like `this` in Java, constructor calls, field accesses, and method invocations. Types are trait types including the top trait Object and object types; note that object types are leaves of any FFMM type hierarchy.

For brevity, we make several assumptions that are easily checked syntactically: (1) every trait or object declaration declares a unique name, (2) every trait or object extends at least one trait, (3) extended traits in every trait or object are different, (4) every field has a unique name in its defining object, (5) no trait nor object declares Object, (6) type hierarchies are acyclic, and (7) every variable in type environments is unique.

3.2 Overloading Rules

In this section, we formally define the overloading rules described in prose in Section 2. The static semantics of FFMM describes how to type-check a given program at compile time. Type-checking a program consists of checking its trait and object declarations and the top-level expression:

Getting visible methods: $\boxed{defined_p(C)\,/inherited_p(C)\,/visible_p(C) = \{\overrightarrow{(m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}, \overrightarrow{x}.\, e)}\}}$

$$
\begin{aligned}
defined_p(C) &= \{(m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}, \overrightarrow{x}.\, e) \mid m(\overrightarrow{x\colon \tau_{\mathrm{p}}})\colon \tau_{\mathrm{r}} = e \in \overrightarrow{md}\} \\
&\quad \text{where } \_\ C\ \_\ \overrightarrow{md}\ \texttt{end} \in p \\
inherited_p(C) &= \{(m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}, \overrightarrow{x}.\, e) \mid (m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau_{\mathrm{r}}, \overrightarrow{x}.\, e) \in visible_p(T), \\
&\qquad\qquad \textit{there is no } \tau' \textit{ such that } (m, \overrightarrow{\tau_{\mathrm{p}}} \to \tau', \_) \in defined_p(C)\} \\
&\quad \text{where } \_\ C\ \texttt{extends}\ \{\overrightarrow{T}\}\ \_ \in p \\
visible_p(C) &= defined_p(C) \cup inherited_p(C)
\end{aligned}
$$

Fig. 3. $visible_p$ function

$$
\text{[T-PROGRAM]}\ \frac{p = \overrightarrow{d}\ e \qquad p \vdash \overrightarrow{d}\ \mathsf{ok} \qquad p; \emptyset \vdash e : \tau}{\vdash p : \tau}
$$

Type-checking a trait, for example, includes checking a set of method declarations visible from the trait:

$$
\text{[T-TRAITDEF]}\ \frac{p \vdash \overrightarrow{T'}\ \mathsf{ok} \qquad p; \texttt{self} : T \vdash \overrightarrow{md}\ \mathsf{ok} \qquad p \vdash validMeth(T)}{p \vdash \texttt{trait}\ T\ \texttt{extends}\ \{\overrightarrow{T'}\}\ \overrightarrow{md}\ \texttt{end}\ \mathsf{ok}}
$$

via the $validMeth$ judgment:

$$
\text{[VALIDMETH]}\ \frac{\begin{array}{l}\forall\{(md, C), (md', C')\} \subseteq visible_p(C^{\mathrm{o}}). \\ \quad md \neq md',\ \text{(not same declaration)} \\ \quad md = m(\overrightarrow{\_ : \tau^a}) : \tau^r\ \_, \qquad md' = m(\overrightarrow{\_ : \tau^{a'}}) : \tau^{r'}\ \_, \\ \quad p \vdash valid(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, visible_p(C^{\mathrm{o}}))\end{array}}{p \vdash validMeth(C^{\mathrm{o}})}
$$

It checks a set of method declarations visible from a trait or object C^{o}, $visible_p(C^{\mathrm{o}})$ presented in Figure 3, to see whether each pair in the set is a valid overloading. The metavariable C ranges over both trait and object names.

A pair of declarations md and md' is a valid overloading if it satisfies one of the overloading rules: Exclusion Rule, Subtype Rule, and Meet Rule. The pair is checked by the $valid$ judgment described in Figure 4. The [VALIDEXC] rule describes the Exclusion Rule. While Fortress allows programmers to declare disjoint types with `excludes` clauses, FFMM does not support such clauses for brevity because they are largely orthogonal to multiple dispatch. Therefore, a pair of method declarations satisfies the Exclusion Rule when they have different number of parameters. The [VALIDSUBTYR] and [VALIDSUBTYL] rules describe the Subtype Rule. If the parameter type of one declaration is a strict subtype of the parameter type of the other declaration, and the return type of the former is a subtype of the return type of the latter, then the pair satisfies the Subtype Rule. Finally, the [VALIDMEET] rule describes the Meet Rule. If there is

$$[\textsc{ValidExc}] \quad \frac{|\overrightarrow{\tau^a}| \neq |\overrightarrow{\tau^{a'}}|}{p \vdash \mathit{valid}(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

$$[\textsc{ValidSubTyR}] \quad \frac{|\overrightarrow{\tau^a}| = |\overrightarrow{\tau^{a'}}| \quad C\ \overrightarrow{\tau^a} \neq C'\ \overrightarrow{\tau^{a'}} \quad p \vdash \overrightarrow{\tau^{a'}} <: \overrightarrow{\tau^a} \quad p \vdash \tau^{r'} <: \tau^r \quad p \vdash C' <: C}{p \vdash \mathit{valid}(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

$$[\textsc{ValidSubTyL}] \quad \frac{|\overrightarrow{\tau^a}| = |\overrightarrow{\tau^{a'}}| \quad C\ \overrightarrow{\tau^a} \neq C'\ \overrightarrow{\tau^{a'}} \quad p \vdash \overrightarrow{\tau^a} <: \overrightarrow{\tau^{a'}} \quad p \vdash \tau^r <: \tau^{r'} \quad p \vdash C <: C'}{p \vdash \mathit{valid}(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

$$[\textsc{ValidMeet}] \quad \frac{\begin{array}{l} l = |\overrightarrow{\tau^a}| = |\overrightarrow{\tau^{a'}}| \quad C\ \overrightarrow{\tau^a} \neq C'\ \overrightarrow{\tau^{a'}} \quad \tau_0^a = C \quad \tau_0^{a'} = C' \\ \exists (m(\overrightarrow{\_ : \tau^{a''}}) : \_, \tau_0^{a''}) \in S. \\ \text{where } (l = |\overrightarrow{\tau^{a''}}|) \ \wedge\ (\forall\, 0 \leq i \leq l.\, p \vdash \mathit{meet}(\{\tau_i^a,\ \tau_i^{a'},\ \tau_i^{a''}\}, \tau_i^{a''})) \end{array}}{p \vdash \mathit{valid}(m, C, \overrightarrow{\tau^a} \to \tau^r, C', \overrightarrow{\tau^{a'}} \to \tau^{r'}, S)}$$

Fig. 4. Overloading rules

Meet type: $\boxed{p \vdash \mathit{meet}(\{\overrightarrow{\tau}\}, \tau)}$

$$[\textsc{Meet}] \quad \frac{\tau' \in \{\overrightarrow{\tau}\} \quad p \vdash \tau' <: \overrightarrow{\tau} \quad p \vdash \cap\{\overrightarrow{\tau}\} <: \tau'}{p \vdash \mathit{meet}(\{\overrightarrow{\tau}\}, \tau')}$$

Intersection type: $\boxed{\tau \cap \tau = \tau}$

$$\tau_1 \cap \tau_2 = \begin{cases} \tau_3 & \text{if } \tau_3 <: \tau_1 \ \wedge\ \tau_3 <: \tau_2 \ \wedge\ \tau_1 \not<: \tau_2 \ \wedge\ \tau_2 \not<: \tau_1 \\ & \quad \wedge\ (\forall \tau_4, (\tau_4 <: \tau_1 \ \wedge\ \tau_4 <: \tau_2) \to \tau_4 <: \tau_3) \\ \tau_1 & \text{if } \tau_1 <: \tau_2 \\ \tau_2 & \text{if } \tau_2 <: \tau_1 \end{cases}$$

Fig. 5. Meet and intersection of types

a disambiguating declaration whose parameter type is the *meet* of the parameter types of the declarations in a pair, the pair satisfies the Meet Rule .

Definitions of the meet and the intersection of types are presented in Figure 5. The meet of a set of types is the most specific type in the set, and the intersection of two types is the greatest lower bound of them. These definitions serve a key role in the Meet Rule. As the `tow` example in Section 1 shows, finding a tie-breaking meet is not trivial. Actually, the bug we found in CFwO was in its definition of the Meet Rule: CFwO incorrectly specifies the definition of the meet type and the Meet Rule for multiple inheritance.

Applicable definitions: $\boxed{applicable_p(m(\overrightarrow{\tau}), \{\overrightarrow{(md, C)}\}) = \{\overrightarrow{(md, C)}\}}$

$$applicable_p(m(\overrightarrow{\tau}), S) = \{(md, C) \mid (md, C) \in S,\ md = m(\overrightarrow{x : \tau'}) : \_,\ p \vdash \overrightarrow{\tau} <: \overrightarrow{\tau'}\}$$

Most specific definitions: $\boxed{mostspecific_p(\{\overrightarrow{(md, C)}\}) = \{\overrightarrow{md}\}}$

$$mostspecific_p(\{\overrightarrow{(md, C)}\}) = \begin{cases} \{md_i\} & \text{if } |\overrightarrow{md}| = n \\ & \overrightarrow{md} = m((\overrightarrow{\_ : \tau^a})_1) : \tau^r_1\ \_ \cdots m((\overrightarrow{\_ : \tau^a})_n) : \tau^r_n\ \_ \\ & (md_i, C_i) \in \{\overrightarrow{(md, C)}\} \\ & \forall\, 1 \le j \le n.\, (p \vdash \overrightarrow{\tau^a}_i <: \overrightarrow{\tau^a}_j \wedge p \vdash C_i <: C_j) \\ \emptyset & \text{Otherwise} \end{cases}$$

Fig. 6. $applicable_p$ and $mostspecific_p$functions

3.3 Overloading Resolution

When several of the overloaded methods are applicable to a particular call, the most specific applicable declaration is selected by the dispatch mechanism. The following rule describes how FFMM evaluates a method invocation at run time:

$$\text{[R-METHOD]}\ \frac{\texttt{object}\ O\ \_\ \texttt{end} \in p \qquad \overrightarrow{type(v')} = \overrightarrow{\tau} \qquad mostspecific_p(applicable_p(m(\overrightarrow{\tau}), visible_p(O))) = \{m(\overrightarrow{x : \_}) : \_ = e\}}{p \vdash E[O(\overrightarrow{v}).m(\overrightarrow{v'})] \longrightarrow E[[O(\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x}]e]}$$

Among all the visible methods from the receiver O, $applicable_p(m(\overrightarrow{\tau}), visible_p(O))$ selects the applicable declarations to a method call of name m with arguments of type $\overrightarrow{\tau}$. Note that because our dispatch mechanism uses symmetric multiple dispatch, it considers the dynamic types of all the arguments equally: $\overrightarrow{type(v')} = \overrightarrow{\tau}$. Finally, among the applicable declarations, select the single most specific declaration via the $mostspecific_p$ function.

The definitions of the $applicable_p$ and $mostspecific_p$ functions are presented in Figure 6. For a given method name m, the dynamic types of all the arguments $\overrightarrow{\tau}$, and a set of visible methods $\{\overrightarrow{(md, C)}\}$, the $applicable_p$ function collects all the methods that have the given name and whose parameter types are super types of the given arguments' dynamic types from the set of visible methods. The $mostspecific_p$ function selects the best method to call from a given set of applicable methods.

When a program is well typed under the static semantics of FFMM, there are no undefined nor ambiguous calls at run time. The type safety of FFMM described in Section 5 guarantees this property. Thanks to the type safety, the $mostspecific_p$ function always picks the single most specific method to call at run time.

```
Definition validmeet' (mn: mname) (tys : list typ) (ty : typ) (mS : mSet) : Prop :=
    exists2 mdt, mdt \in mS &
            ((tys = (getartys mdt)) ∧ (ty = (snd mdt)) ∧ (mn = (getmname mdt))).
Inductive valid (mdt1 mdt2 : mdttype) (mS : mSet) : Prop :=
    | valid_same :
        (getmid mdt1) = (getmid mdt2) →
        (snd mdt1) = (snd mdt2) →
        valid mdt1 mdt2 mS
    | valid_diff_name :
        (getmname mdt1) ≠ (getmname mdt2) →
        valid mdt1 mdt2 mS
    | valid_diff_arg_len :
        (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
        (getmname mdt1) = (getmname mdt2) →
        (getenvlen mdt1) ≠ (getenvlen mdt2) →
        valid mdt1 mdt2 mS
    | valid_sub_ty_r :
        (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
        (getartys mdt1) ≠ (getartys mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
        (getmname mdt1) = (getmname mdt2) →
        (getenvlen mdt1) = (getenvlen mdt2) →
        sub_tys (getartys mdt2) (getartys mdt1) →
        sub_ty (getrty mdt2) (getrty mdt1) →
        sub_ty (snd mdt2) (snd mdt1) →
        valid mdt1 mdt2 mS
    | valid_sub_ty_l :
        (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
        (getartys mdt1) ≠ (getartys mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
        (getmname mdt1) = (getmname mdt2) →
        (getenvlen mdt1) = (getenvlen mdt2) →
        sub_tys (getartys mdt1) (getartys mdt2) →
        sub_ty (getrty mdt1) (getrty mdt2) →
        sub_ty (snd mdt1) (snd mdt2) →
        valid mdt1 mdt2 mS
    | valid_meet : ∀ tys ty,
        (getmid mdt1) ≠ (getmid mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
        (getartys mdt1) ≠ (getartys mdt2) ∨ (snd mdt1) ≠ (snd mdt2) →
        (getmname mdt1) = (getmname mdt2) →
        (getenvlen mdt1) = (getenvlen mdt2) →
        ~ (sub_tys (getartys mdt1) (getartys mdt2)) →
        ~ (sub_tys (getartys mdt2) (getartys mdt1)) →
        is_tys (getartys mdt1) (getartys mdt2) tys →
        is_ty (snd mdt1) (snd mdt2) ty →
        validmeet' (getmname mdt1) tys ty mS→
        valid mdt1 mdt2 mS.
```

Fig. 7. Overloading rules in Coq

4 FFMM in COQ

To mechanically prove the type safety of FFMM in Section 5, we describe our implementation of FFMM using COQ 8.3 in this section. Our implementation is largely based on Cast-Free Featherweight Java (CFFJ) by Fraine *et al.* [18]. Among others, the `Metatheory` library in CFFJ provides auxiliary constructs and properties of atoms [2] and environments including membership tests, accessors, and uniqueness guarantee. For our convenience, we extend the library with utility functions mostly for list manipulation. The full implementation is available online [20].

We implement the seven assumptions described in Section 3.1 in three ways:

- The uniqueness assumptions are implemented by the existing `Metatheory` library: (1), (4), and (7)
- We extend the `Metatheory` library to implement the assumptions on traits and objects: (2) and (3)
- To separate the concerns of the well-formed programs assumptions, we use a module system of COQ [14, Chapter 5]: (5) and (6)

While the COQ implementation of FFMM is very close the FFMM calculus, there are small differences in the implementation:

1. Unlike Java-like languages which provide only classes, FFMM provides both traits and objects as we discuss in Section 3. While the calculus does not distinguish between traits and objects by using the metavariable C in most cases, the implementation maintains two separate class tables for traits and objects. With two class tables, we could reuse the existing `Metatheory` library instead of forking a variant of it to handle both traits and objects in one class table.
2. While the calculus identifies each method by its name and parameter types, the implementation introduces a unique identity of type `nat` for each method. With the unique identity for every method, we could again reuse the existing `Metatheory` library as it is instead of forking a variant of it to use a pair of method name and parameter types as keys of environments.
3. As the [VALIDMETH] in Section 3 describes, the calculus does not check the overloading rules for a pair of method declarations if the pair is the same method or the pair has different names. However, for simplicity, the implementation checks the overloading rules for such declarations as Figure 7 illustrates. Unlike the overloading rules of the calculus in Figure 4, the implementation includes two extra cases: valid_same and valid_diff_name. The valid_same constructor specifies that if we check the overloading rules with a single method declaration and itself, the rules are vacuously satisfied. The valid_diff_name constructor specifies that if we check the overloading rules with a pair of method declarations with different names, again the overloading rules hold vacuously.
4. While the overloading rules in the calculus are not disjoint, its corresponding implementation is. The [VALIDMEET] rule in the calculus could be satisfied by a pair of method declarations whose parameter types and return types are in the subtype relation. However, the valid_meet case is not satisfied by such a pair. To make the

implementation of the overloading rules deterministic, the valid_meet constructor specifies that two method declarations are not in the subtype relation:

```
~ (sub_tys (getartys mdt1) (getartys mdt2)) →
~ (sub_tys (getartys mdt2) (getartys mdt1)) →
```

Because the differences in the calculus and the implementation are minor implementation details, we believe that we faithfully implement FFMM in Coq.

5 Type Safety Proof

We prove the type safety of the FFMM calculus in Coq. Among approximately 150 facts, lemmas, and theorems in our proof, we describe only those that are closely related to multiple dispatch and multiple inheritance.

First, the following lemma guarantees that, in a well-typed program, every non-empty set of applicable methods always includes the most specific method:

Lemma 1. *Suppose that* p *is well typed. If* $applicable_p(m(\overrightarrow{\tau}), visible_p(C))=\{\overrightarrow{md, C}\}$ *and* $\{\overrightarrow{md, C}\} \neq \emptyset$*, then there exists* md' *such that* $mostspecific_p(\{\overrightarrow{md, C}\}) = \{md'\}$.

It implies that there are *no ambiguous method calls* at run time in FFMM. In a well-typed program, each method call has at least one applicable method. In other words, the set of applicable methods to a call is not empty as guaranteed by the typing rule for method invocations as follows:

$$[\text{T-Method}]\ \frac{p; \Gamma \vdash e_o : \tau_o \qquad p; \Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau} \qquad mostspecific_p(applicable_p(m(\overrightarrow{\tau}), visible_p(\tau_o))) = \{m(\_) : \tau^r \_\}}{p; \Gamma \vdash e_o.m(\overrightarrow{e}) : \tau^r}$$

It implies that there are *no undefined method calls* at run time in FFMM. This lemma plays an important role in the proofs of the following two lemmas:

Lemma 2. *Let* p *be well typed. If* $mostspecific_p(applicable_p(m(\overrightarrow{\tau'}), visible_p(C))) = \{m(\overrightarrow{x' : \tau^{a'}}) : \tau^{r\prime} = e'\}$ *and* $p \vdash \overrightarrow{\tau} <: \overrightarrow{\tau'}$ *for some* $\overrightarrow{\tau}$*, then there exists* $m(\overrightarrow{x : \tau^a}) : \tau^r = e$ *such that* $mostspecific_p(applicable_p(m(\overrightarrow{\tau}), visible_p(C))) = \{m(\overrightarrow{x : \tau^a}) : \tau^r = e\}$ *and* $p \vdash \tau^r <: \tau^{r\prime}$.

Lemma 3. *Let* p *be well typed. If* $mostspecific_p(applicable_p(m(\overrightarrow{\tau}), visible_p(C'))) = \{m(\overrightarrow{x' : \tau^{a'}}) : \tau^{r\prime} = e'\}$ *and* $p \vdash C <: C'$ *for some* C*, then there exists* $m(\overrightarrow{x : \tau^a}) : \tau^r = e$ *such that* $mostspecific_p(applicable_p(m(\overrightarrow{\tau}), visible_p(C))) = \{m(\overrightarrow{x : \tau^a}) : \tau^r = e\}$ *and* $p \vdash \tau^r <: \tau^{r\prime}$.

These lemmas state that dynamically selecting a method that is more specific than the statically chosen method is type safe. They serve an important role in proving that term substitutions preserve typing:

Table 1. COQ mechanization of CFFJ, FBCF, and FFMM

Language	Metatheory		Calculus	Type Safety Proof		Total	
	Spec	Proofs	Spec	Spec	Proofs	Spec	Proofs
CFFJ	114	158	164	249	338	527	496
FBCF	114	158	226	233	348	573	506
FFMM	136	203	402	742	1786	1280	1989

Lemma 4. *Suppose that p is well typed. If $p; \Gamma\overrightarrow{x:\tau} \vdash e : \tau$ and $p; \Gamma \vdash \overrightarrow{e'} : \overrightarrow{\tau'}$, and $p \vdash \overrightarrow{\tau'} <: \overrightarrow{\tau}$, then $p; \Gamma \vdash [\overrightarrow{e'/x}]e : \tau'$ for some τ' such that $p \vdash \tau' <: \tau$.*

Finally, the type safety proof of FFMM follows the traditional technique:

Theorem 1 (Progress). *Suppose that p is well typed. If $p; \emptyset \vdash e : \tau$, then e is a value or there exists some e' such that $p \vdash e \longrightarrow e'$.*

The Progress theorem is proved by induction on the derivation of $p; \emptyset \vdash e : \tau$. The most interesting part is the [T-METHOD] case where we find a witness e' using the previous lemmas.

Theorem 2 (Preservation). *Suppose that p is well typed. If $p ; \Gamma \vdash e : \tau$ and $p \vdash e \longrightarrow e'$, then $p; \Gamma \vdash e' : \tau'$ where $p \vdash \tau' <: \tau$.*

The type safety theorem is immediate from Theorem 1 and Theorem 2:

Theorem 3 (Type Safety). *Suppose that p is well-typed. If $p; \emptyset \vdash e : \tau$ and $p \vdash e \longrightarrow^* v$, then $p; \emptyset \vdash v : \tau'$ and $p \vdash \tau' <: \tau$.*

The full proof of all the facts, lemmas, and theorems in COQ is available online [20].

6 Lessons

In this section, we share our experience and lessons from mechanizing the type soundness of FFMM.

6.1 Extensibility of COQ Mechanization

Before FFMM, we mechanized Featherweight Basic Core Fortress (FBCF) [22], a very small core of the Fortress programming language, in COQ. It supports both traits and objects like FFMM, but it does not provide multiple dispatch nor multiple inheritance. Its mechanization heavily relies on the implementation of CFFJ. Table 1 compares the line numbers of the COQ implementations of CFFJ, FBCF, and FFMM. While the size of the FBCF implementation is similar to that of the CFFJ implementation, the size of the FFMM implementation is almost three times bigger than them.

While FBCF provides method overriding and single inheritance, FFMM supports method overloading and multiple inheritance. Similarly to CFFJ, FBCF uses the $mtype_p$ and $mbody_p$ functions for method lookup: it traverses up the type hierarchy one by one until it finds the intended method. On the contrary, FFMM uses the $visible_p$ function for method lookup: it collects all the visible methods first instead of traversing up the type hierarchy.

We observed that adding multiple inheritance to FFMM was much easier and natural than adding it to FBCF. We first tried to extend FBCF by generalizing the $mtype_p$ and $mbody_p$ functions to support multiple inheritance, but it became easily exponential. Extending FFMM with multiple inheritance amounts to collecting all the super types before collecting the visible methods via the $visible_p$ function, which does not require much code changes.

6.2 Witness Finding

When a rule in FFMM is not algorithmic, finding a witness to mechanize the rule in Coq could be not trivial. For example, in the following subtype transitivity rule:

$$[\text{S-TRANS}]\ \frac{p \vdash \tau_1 <: \tau_2 \qquad p \vdash \tau_2 <: \tau_3}{p \vdash \tau_1 <: \tau_3}$$

the premise uses τ_2 that does not appear in the conclusion. Therefore, when we prove a statement including subtype relations, we should be able to find a witness for the [S-TRANS] case. Consider the following fact:

```
Fact sub_ty_implies_visible_super_set: ∀ ty ty' mS mS',
  sub_ty ty ty' →
  visible ty' mS' →
  visible ty mS →
  (∀ mdt, mdt \in mS' → mdt \in mS).
```

which states that "If *ty* is a subtype of *ty'*, $visible_p(ty') = mS'$, and $visible_p(ty) = mS$, then *mS'* is a subset of *mS*." Proving the above fact by induction on the derivation of **sub_ty** *ty ty'* fails to find a witness for the [S-TRANS] case: while Coq has to find a witness of $visible_p(ty'')$ for some *ty"* that is a subtype of *ty'* and a supertype of *ty*, two method sets, *mS'* and *mS*, are already bound to two types, *ty'* and *ty*, respectively. Moreover, the fact statement does not specify how to find a set of visible methods for a certain type that is not mentioned in the statement such as *ty"*.

Instead, we need to restate the above fact as follows:

```
Fact sub_ty_implies_visible_super_set: ∀ ty ty' mS',
  sub_ty ty ty' →
  visible ty' mS' →
  exists2 mS, visible ty mS & (∀ mdt, mdt \in mS' → mdt \in mS).
```

By using `exists2` in the induction hypothesis, Coq can find a witness to prove the [S-TRANS] case.

7 Related Work

Several object-oriented languages provide multiple dispatch. As we discussed in Section 1, there are two camps in multiple dispatch: asymmetric multiple dispatch and symmetric multiple dispatch. Languages supporting asymmetric multiple dispatch such as

CLOS [9] and Dylan [1] distinguish method arguments to eliminate the possibility of ambiguous method calls. However, languages with symmetric multiple dispatch such as Nice [10] and Fortress [3] treat all the arguments equally, and they provide some restrictions to guarantee no ambiguous calls at run time.

Some languages support symmetric multiple dispatch with static rules on overloaded method declarations. Castagna *et al.* [12] proposed the $\lambda\&$-calculus, an extension of the typed lambda calculus with overloaded functions, and presented constraints to ensure that for each call site, there exists a unique best method to call at run time. Similarly, the Fortress team [4] proposed static rules on overloaded methods in the context of the Fortress type system. Millstein and Chambers designed the Dubious [24] language and restrictions to ensure the *modular* type safety in the presence of symmetric multiple dispatch.

Researchers have proposed extensions of the Java programming language with symmetric multiple dispatch. Clifton *et al.* [13] presented MultiJava which adds symmetric multiple dispatch and open classes to Java. Lorenzo *et al.* [7,8] proposed Featherweight Java with Multi-methods (FJM) and proved its type soundness. Lievens and Harrison [23] proposed a very similar approach to FJM but they included casting expressions that are omitted in FJM. None of these extensions support multiple inheritance.

Type safety proofs of several languages are mechanized in COQ. Dubois [17] proved type soundness of ML [25] using COQ, Fraine *et al.* [18] proved the type safety of CFFJ, Delaware *et al.* [16] verified the type soundness of Lightweight Feature Java, a subset of Java extended with support for features, and Cremet and Altherr [6,15] mechanized the type safety of FGJ_Ω, an extension of FGJ with variables representing type constructors. However, none of these mechanizations of Java-like languages provide multiple dispatch nor multiple inheritance.

8 Conclusion and Future Work

We present a core calculus for the Fortress programming language with multiple dispatch and multiple inheritance. The calculus formally specifies the static restrictions on valid overloaded method declarations. For a well-typed program, which satisfies the overloading rules statically, there are no undefined nor ambiguous calls at run time. We mechanize the calculus and prove its type safety in COQ. As far as we know, our work is the first mechanized calculus of multiple dispatch in the presence of multiple inheritance, and we believe that our work is adaptable to any object-oriented languages with multiple dispatch and multiple inheritance.

We are planning to extend the calculus with more features in Fortress and mechanize the extended calculus and its type safety proof. First, we are planning to add `excludes` clauses to the calculus so that the Exclusion Rule can allow more methods as valid overloadings. Secondly, using the Fortress team's recent work on valid overloadings of parametrically polymorphic methods [5], we will extend the calculus to permit overloadings on generic methods. Finally, we will support the Fortress module system so that the calculus can faithfully capture the core expressive power of the Fortress overloading mechanism.

Acknowledgments. This work is supported in part by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology (MEST) / National Research Foundation of Korea(NRF) (Grant 2011-0000974).

References

1. Dylan, http://www.opendylan.org/
2. Metatheory Library: Atom, http://www.cis.upenn.edu/ plclub/popl08-tutorial/code/coqdoc/Atom.html
3. 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)
4. Allen, E., Hallett, J.J., Luchangco, V., Ryu, S., Steele Jr., G.L.: Modular Multiple Dispatch with Multiple Inheritance. In: Proceedings of the 2007 ACM Symposium on Applied Computing, New York, NY, USA, pp. 1117–1121. ACM (2007)
5. Allen, E., Hilburn, J., Kilpatrick, S., Ryu, S., Chase, D., Luchangco, V., Steele Jr., G.L.: Type-checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance. In: Proceedings of the 26th ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications. ACM (2011)
6. Altherr, P., Cremet, V.: Adding Type Constructor Parameterization to Java. In: Formal Techniques for Java-like Programs (2007)
7. Bettini, L., Capecchi, S., Venneri, B.: Featherweight Java with Multi-methods. In: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, New York, NY, USA, pp. 83–92. ACM (2007)
8. Bettini, L., Capecchi, S., Venneri, B.: Featherweight Java with Dynamic and Static Overloading. Science of Computer Programming 74, 261–278 (2009)
9. Bobrow, D.G., DiMichiel, L.G., Gabriel, R.P., Keene, S.E., Kiczales, G., Moon, D.A.: Common Lisp Object System Specification. ACM SIGPLAN Notices, 23 (September 1988)
10. Bonniot, D., Keller, B., Barber, F.: The Nice user's manual (2003), http://nice.sourceforge.net/NiceManual.pdf
11. Bracha, G., Steele, G., Joy, B., Gosling, J.: Java™ Language Specification, 3rd edn. Java Series. Addison-Wesley Professional (July 2005)
12. Castagna, G., Ghelli, G., Longo, G.: A Calculus for Overloaded Functions with Subtyping. SIGPLAN Lisp Pointers 1, 182–192 (1992)
13. Clifton, C., Leavens, G.T., Chambers, C., Millstein, T.: MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In: Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, pp. 130–145. ACM (2000)
14. The Coq Development Team. The Coq Proof Assistant, http://coq.inria.fr/
15. Cremet, V., Altherr, P.: FGJ-ω in Coq (2007), http://lamp.epfl.ch/~cremet/FGJ-omega
16. Delaware, B., Cook, W.R., Batory, D.: Fitting the Pieces Together: a Machine-checked Model of Safe Composition. In: Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC/FSE 2009, New York, NY, USA, pp. 243–252. ACM (2009)
17. Dubois, C.: Proving ML Type Soundness within Coq. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 126–144. Springer, Heidelberg (2000)
18. De Fraine, B., Ernst, E., Südholt, M.: Cast-Free Featherweight Java (2008), http://soft.vub.ac.be/~bdefrain/featherj

19. Igarashi, A., Pierce, B., Wadler, P.: Featherweight Java: A Minimal Core Calculus for Java and GJ. In: Meissner, L. (ed.) Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA 1999), vol. 34(10), pp. 132–146. NY (1999)
20. Kim, J.: FFMM in Coq (2011), http://plrg.kaist.ac.kr/_media/research/software/ffmm_in_coq.tar.gz
21. Kim, J., Ryu, S.: FFMM: Featherweight Fortress with Multiple Dispatch and Multiple Inheritance. Technical report, KAIST (June 2011)
22. Kim, J., Ryu, S.: Coq Mechanization of Featherweight Basic Core Fortress for Type Soundness. Technical Report ROSAEC-2011-011, KAIST (May 2011)
23. Lievens, D., Harrison, W.: Symmetric Encapsulated Multi-methods to Abstract over Application Structure. In: Proceedings of the 2009 ACM Symposium on Applied Computing, New York, NY, USA, pp. 1873–1880. ACM (2009)
24. Millstein, T., Chambers, C.: Modular Statically Typed Multimethods. In: Information and Computation, pp. 279–303 (2002)
25. Milner, R., Tofte, M., Harper, R., MacQueen, D.: The Definition of Standard ML (Revised). The MIT Press (1997)
26. Muschevici, R., Potanin, A., Tempero, E., Noble, J.: Multiple Dispatch in Practice. In: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications, pp. 563–582. ACM (2008)
27. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-Step Guide, 2nd edn. Artima Inc. (2011)
28. 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)
29. Stroustrup, B.: The C++ Programming Language. Addison-Wesley (1985)