석사 학위논문
Master’s Thesis

Coq을 이용한 FFMM의 타입 안전성 증명

Proving FFMM Type Safety Using COQ

김 지응 (金 知 應 Kim, Jieung)
전산학과
Department of Computer Science

KAIST

2011
Coq을 이용한 FFMM의 타입 안전성 증명

Proving FFMM Type Safety Using Coq
Proving FFMM Type Safety Using COQ

Advisor : Professor Sukyoung Ryu

by
Kim, Jieung
Department of Computer Science
KAIST

A thesis submitted to the faculty of KAIST in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Computer Science. The study was conducted in accordance with Code of Research Ethics¹.

2011. 06. 09.
Approved by
Professor Sukyoung Ryu
[Advisor]

¹Declaration of Ethical Conduct in Research: I, as a graduate student of KAIST, hereby declare that I have not committed any acts that may damage the credibility of my research. These include, but are not limited to: falsification, thesis written by someone else, distortion of research findings or plagiarism. I affirm that my thesis contains honest conclusions based on my own careful research under the guidance of my thesis advisor.
Coq을 이용한 FFMM의 타입 안전성 증명

김 지 응

위 논문은 한국과학기술원 석사학위논문으로 학위논문심사위원회에서 심사 통과하였음.

2011년 06월 09일

심사위원장 류석영 (인)
심사위원 최광무 (인)
심사위원 한태숙 (인)
ABSTRACT

In this paper, we design a core calculus of the Fortress programming language, Featherweight Fortress with Multiple Dispatch and Multiple Inheritance (FFMM), which provides both multiple dispatch and multiple inheritance, and mechanize its type safety proof using CoQ. Multiple dispatch allows method selection among overloaded methods at run time based on dynamic types of more than one method arguments, and multiple inheritance allows a type to have more than one super types. Both of them give high expressive power to programming languages. However, languages that provide multiple dispatch and multiple inheritance need static restrictions to prevent undefined method calls and ambiguous method calls at run time. While previous work proposed static restrictions to guarantee no undefined method calls and no ambiguous method calls at run time, they are not closely tied to a particular programming language. Therefore, we devise a formal calculus that reflects static restrictions for safe multiple dispatch and multiple inheritance. On top of that, to rigorously show the type safety of our calculus, we mechanize the calculus and its type safety proof using CoQ.
## Contents

Abstract ................................................................. i  
Contents .............................................................. ii  
List of Tables ......................................................... iv  
List of Figures ........................................................ v  

Chapter 1. Introduction .............................................. 1  

Chapter 2. Background .............................................. 2  
  2.1 Method Overloading and Dispatch Mechanism ............... 2  
  2.2 Multiple Dispatch and Multiple Inheritance .............. 3  
  2.3 Overloading Rules .............................................. 4  
  2.4 Mechanization .................................................. 5  

Chapter 3. Featherweight Fortress with Multiple Dispatch and Multiple Inheritance ......................... 6  
  3.1 FFMM Overview .................................................. 6  
  3.2 Syntax of FFMM .................................................. 6  
  3.3 Auxiliary Functions ............................................ 7  
  3.4 Static Semantics ............................................... 8  
  3.5 Dynamic Semantics ............................................ 13  

Chapter 4. FFMM in CoQ ............................................ 14  
  4.1 Assumptions of FFMM in CoQ .................................. 14  
  4.2 Syntax and Semantics of FFMM in CoQ ...................... 15  
  4.3 FFMM in CoQ Summary ......................................... 17  

Chapter 5. Type Safety Proof in CoQ .......................... 18  
  5.1 Properties ....................................................... 18  
  5.2 Proof Summary ................................................ 21  

Chapter 6. Comparing Existing CoQ Implementations ..... 22  
  6.1 CFFJ, FBCF, and FFMM ......................................... 22  
  6.2 CFFJ, LJ, and FGJΩ ............................................ 23  

Chapter 7. Related Work .......................................... 24
## List of Tables

4.1 CoQ libraries to define FFMM calculus ............................................. 17
5.1 CoQ libraries to prove FFMM type safety ............................................. 21
6.1 CFFJ, FBCF, and FFMM in CoQ ....................................................... 23
6.2 CFFJ, LJ, and FGJ\(Ω\) in CoQ ....................................................... 23
List of Figures

2.1 Ambiguity with multiple dispatch ........................................... 3
2.2 Ambiguity with multiple inheritance ........................................ 3
2.3 Ambiguity example to show the necessity of the meet rule ............... 4

3.1 applicable_p and mostspecific_p functions ................................. 8
3.2 A class hierarchy for examples of method selection ....................... 8
3.3 Intersection type and meet rule ............................................. 10
3.4 Overloading rules in FFMM ................................................... 11
3.5 Expression typing rules ..................................................... 12
3.6 Definition typing rules ...................................................... 12
3.7 Dynamic semantics .......................................................... 13

4.1 HYPS module in CoQ .......................................................... 15
4.2 Overloading rules in CoQ .................................................... 16

5.1 Preservation theorem proof in CoQ ......................................... 19
5.2 Progress theorem proof in CoQ ............................................. 20
Chapter 1. Introduction

Most object oriented languages support method overloading. Since method overloading allows multiple method declarations with the same name, programming languages that have method overloading should have a method selection mechanism to pick a method to be called among several candidates. If one method is selected according to a particular process, we call it a dispatch mechanism. There are several dispatch mechanisms to handle method overloading. Among those mechanisms, multiple dispatch gives high expressive power to programming languages, and it gives a natural solution to several problems that inhere in single dispatch [8, 5, 9, 14]. For example, multiple dispatch provides a natural solution to some drawbacks of single dispatch such as “binary methods” or “visitor pattern”.

Multiple inheritance allows the type to reuse code in its multiple super types. It permits a more flexible type hierarchy of the program than single inheritance, and therefore, as with multiple dispatch, it provides high expressive power to the programming language.

However, when embedding two features in a particular language, if the programming language does not have any restrictions for method overloading, problems that are related to method calls are raised. Among those problems, two main problems related to method calls are an undefined method call error and an ambiguous method call error. Therefore, to rule out those problems, the language should have static restrictions that can check overloaded methods.

The Fortress programming language [32] is a new object oriented language, and it has multiple dispatch and multiple inheritance in its features [10]. The Fortress team [9] has proposed static restrictions to guarantee no undefined method calls and no ambiguous method calls for all method invocations at run time. However, those restrictions are not closely tied to a particular programming language.

Therefore, in this paper, we design a formal calculus, a subset calculus of the Fortress language, by reflecting those restrictions properly. On top of that, to check whether our calculus satisfies desired properties or not, we mechanize the syntax and semantics of our calculus and its type safety proof using a proof assistant tool. For mechanization, we chose the famous proof assistant tool, Coq [30], and used the latest stable version of Coq, version 8.3.

The remainder of this paper is organized as follows. In Chapter 2, we discuss basic background information related to our work, including overloading rules for safe multiple dispatch and multiple inheritance. Chapter 3 presents the calculus that we designed, Featherweight Fortress with Multiple Dispatch and Multiple Inheritance (FFMM), and illustrates every detail of our calculus, especially how the overloading rules in Chapter 2 are injected in our calculus. In Chapter 4, we explain some aspects of our Coq implementation, especially those that are related to the syntax and semantics of FFMM, and Chapter 5 illustrates the type safety proof of our calculus in Coq. Chapter 6 compares existing Coq implementations for the type safety proof of several small object oriented languages, and we discuss some related work in Chapter 7. Chapter 8 concludes with several directions for future work.
Chapter 2. Background

2.1 Method Overloading and Dispatch Mechanism

Method overloading allows many methods, that have the same name, declared within the same scope. It is also called ad hoc polymorphism, in contrast to parametric polymorphism that allows for methods whose code works uniformly on arguments of different types [1]. With method overloading, given a method call, the language must select a method to be called. If a single definition is to be chosen and invoked, then we call these determination processes as dispatch mechanisms. There are numerous dispatch mechanisms for method overloading. Among those mechanisms, static dispatch and dynamic dispatch are categorized by the way of using type information of method arguments. When only static type information of all method arguments is used in method selection, we call it static dispatch. Therefore, using static dispatch, method selection is performed at some time prior to programming execution. On the other hand, if the programming language looks up dynamic (or run time) type information of method arguments for method selection, we call it as dynamic dispatch. Several object oriented languages such as JavaTM [2], C++ [3], C# [31], Scala [4], and Fortress [32] allow multiple definitions that have the same method name (method overloading), and all those languages use run time type information of method arguments to select the most specific method to be called at run time (dynamic dispatch).

Dynamic dispatch can be categorized as two sub terms: single dispatch and multiple dispatch. To select a method to be called, single dispatch refers to the run time type information of the first method argument (often called the “receiver” in Java) and the static type information of all other method arguments. For instance, JavaTM, C++, C#, SIMULA [34], and Smalltalk [35] use single dispatch for their method selection. However, this single dispatch sometimes returns undesirable results even if it is clear that more specific methods exist.

For example, with a such Java program:

```java
class Car {
    void collide (Car c) {print('Car hits Car');}
    ...
}

class CampingCar extends Car {
    void collide (Car c) {print('Camping car this car');}
    void collide (CampingCar) {print('Camping car hits camping car');}
    ...
}

Car cc1 = new CampingCar();
Car cc2 = new CampingCar();
c1.c collide(cc2);
```

most people want to get the message “Camping car hits camping car” as a result. However, according to the single dispatch policy, this Java program returns “Camping car hits car.”

On the other hand, multiple dispatch (or multi-methods) uses run time type information of more than one method arguments (generally, all arguments), to select the most suitable method to be called. If the program
language uses multiple dispatch, we can get the desired result without additional treatments in the same example that we present above.

2.2 Multiple Dispatch and Multiple Inheritance

Multiple dispatch uses the dynamic information of more than one arguments to select the most specific method as we mentioned in Section 2.1, and multiple inheritance permits a type to have more than one super types. Both of them provide high expressive power and flexibility to programming languages. However, benefits of multiple dispatch and multiple inheritance are not a panacea. Both multiple dispatch and multiple inheritance introduce the possibility of an ambiguous method call that cannot be resolved at run time. The following examples show several scenarios in which an ambiguous method call at run time can happen.

Figure 2.1 shows the possibility of an ambiguous method call due to multiple dispatch. When we want to call a `collide` method with two arguments that have a `CampingCar` type at run time, the run time engine cannot determine a method to be called. The `collide(Car c, CampingCar cc)` is not more specific than the `collide(CampingCar cc, Car c)` and the `collide(CampingCar cc, Car c)` is not more specific than the `collide(Car c, CampingCar cc)` with the given method call. Some programming languages are breaking the symmetric treatment of their method arguments to eliminate this problem. For instance, CLOS [6] and Dylan [33] are giving a precedence in argument positions according to the specific order (mostly left to right). By doing that, the `collide(CampingCar cc, Car c)` precedes the `collide(Car c, CampingCar cc)` in method selection, and no ambiguity occurs when we call a `collide` method with two arguments that have a `CampingCar` type at run time. However, this approach places too much burden on programmers. Programmers have to determine argument positions when they create overloaded methods or methods that can be overloaded.

Figure 2.2 shows the possibility of an ambiguous method call due to multiple inheritance. When we want to call a `lightOn` method with one argument which has a `CampingCar` type at run time, the run time engine cannot determine which of two methods have to be selected. Some languages such as CLOS and Dylan are
solving this problem by linearizing the inheritance hierarchy. In those languages, when we assume that a Car type is prior to a CampingTrailer type for a CampingCar type, the lightOn(Car c) is more specific than the lightOn(CampingTrailer ct) with a method call using an argument that has a CampingCar type at run time. However, we do not care of this approach to make natural semantics as possible as we can.

To solve ambiguity problems in Figures 2.1 and 2.2 while keeping symmetric treatments for method arguments and super types, the static type-checker has to verify that every method has a more specific method than itself. For example, the static type-checker argues that programmers should create a method collide(SportsCar sc1, SportsCar sc2) in the example of Figure 2.1 and a method lightOn(CampingCar cc1, CampingCar cc2) in the example of Figure 2.2.

However, this restriction is not sufficient to prevent all possibilities of ambiguous method calls at run time. In Figure 2.3, every method has a more specific method than itself. The method collide(CampingCar cc, CampingCar cc) is more specific than the other two methods, and there are no methods that are more specific than the method collide(CampingCar cc, CampingCar cc). However, the method set in Figure 2.3 is not safe from ambiguous method calls at runtime since programs might want to call a collide method with two arguments that have a Car type at run time. In such a case, the method collide(CampingCar cc, CampingCar cc) is not applicable, and the other two methods are not more specific than collide(Vehicle v, Car c),collide(Car c, Vehicle v),and collide(CampingCar cc, CampingCar cc) for the given method call. Therefore, we need more powerful restrictions to rule out ambiguity; every method pair with the same name should be disjointed or has to have a method that is more specific than both whose parameter type is an intersection type (see Figure 3.3) of the parameter types of two methods.

2.3 Overloading Rules

As we discussed in Section 2.2, with multiple dispatch and multiple inheritance, the programming language has to have a set or restrictions for overloaded methods to prevent ambiguous method calls at run time. The Fortress team [9] designed a set of static overloading rules and proved that the rules guarantee the existence of the most specific method for each method call at run time. In this model, $P \preceq Q$ means $P$ is a subtype of $Q$ and $P \prec Q$ means that $P$ is a subtype of $Q$ and $P \neq Q$. They use $P \cap Q$ to represent an intersection type of $P$ and $Q$, (i.e., $P \cap Q$ is a greatest lower bound of $P$ and $Q$ in the type hierarchy of the program).

Three rules are as follows:

1. **The Exclusion Rule**: If $P$ and $Q$ are disjoint types then $f(P)$ and $f(Q)$ are a valid overloading.
2. **The Subtype Rule**: If $P \prec Q$ and $U \preceq V$, then $f(P) : U$ and $f(Q) : V$ are a valid overloading.

3. **The Meet Rule**: If neither $P$ nor $Q$ is a subtype of the other, then $f(P)$ and $f(Q)$ are a valid overloading if $f(P \cap Q)$ declaration exists.

We can argue that a pair of declarations is a valid overloading if it satisfies one of the rules. The first condition is trivial and quite natural. If two methods have distinct argument types, it is clear that both of them cannot be applicable at the same time. The second condition is related to a specialization of method selection at runtime. This rule needs to guarantee type safety of the programming language; if one method is selected at static time and if a more specialized method selected at runtime method call, its return type should be a subtype of a return type of a statically selected method. The third one is a powerful and crucial condition to rule out all possible ambiguities at runtime method calls. By using this rule, the language can check the ambiguity problem in Figure 2.3

### 2.4 Mechanization

Section 2.3 describes a set of rules using a natural language with few mathematical notations (we will refer to this language as an *informal description*). They proved that no undefined method call and no ambiguous method calls at run time is guaranteed with a certified overloaded method set by those rules. However, the rules are specified independently for the underlying languages, Fortress. Therefore, this informal description still has a gap between the rules and a particular programming language. Except an informal description, we can use two more ways to formally embed those static rules in a particular programming language: *formal calculus* and *mechanization*.

First, we can represent those restrictions using inference rules in a formal defined calculus and prove its type safety by hand. The Fortress team [10, Appendix A.2] made a formal calculus, Core Fortress with Overloading (CFWO) based on their previous approach (an informal description). However, they did not prove the type safety of their calculus and we found that CFWO has a bug. More precisely, the CFWO static type-checker cannot check every possible ambiguity problem, especially related to the problem in Figure 2.3.

Second, we can implement the formal calculus that has some restrictions for overloaded methods using a proof assistant tool and mechanize its type safety proof. Through our efforts, we could not find a mechanized calculus that has multiple dispatch and multiple inheritance. We used Coq [30], one of the proof assistant tools that are available to mechanize the type safety proof of the programming language. Coq is an interactive proof assistant tool, and some researchers [24, 21, 25] have mechanized type safety proofs of their calculi using Coq.
Chapter 3. Featherweight Fortress with Multiple Dispatch and Multiple Inheritance

3.1 FFMM Overview

We made a core subset of the Fortress programming language that has multiple dispatch and multiple inheritance, and we call it Featherweight Fortress with Multiple Dispatch and Multiple Inheritance (FFMM). As we mentioned in Section 2.4, the Fortress team already proposed a formal calculus that includes multiple dispatch and multiple inheritance, and they named the calculus CF\textsubscript{WO}. In contrast to CF\textsubscript{WO}, FFMM does not support generic types nor top-level functions. We remove generic types in FFMM for simplicity and leave it for our future work. In addition, we remove top-level functions since they are orthogonal to the overloading rules in Section 2.3.

3.2 Syntax of FFMM

The syntax of FFMM allows only a small subset of the Fortress language to be formalized. The syntax of FFMM is as follows:

\begin{align*}
\tau & ::= T \quad \text{type} \\
& \quad | \quad O \\
\omega & ::= x \quad \text{expression} \\
& \quad | \quad \text{self} \\
& \quad | \quad O(\overrightarrow{v}) \\
& \quad | \quad e.f \\
& \quad | \quad e.m(\overrightarrow{v}) \\
md & ::= m(\overrightarrow{x}; \tau) : \tau = e \quad \text{method definition} \\
d & ::= \text{trait } T \text{ extends } \{ \overrightarrow{T} \} \overrightarrow{md} \text{ end} \quad \text{trait or object definition} \\
& \quad | \quad \text{object } O(\overrightarrow{f}; \tau) \text{ extends } \{ \overrightarrow{T} \} \overrightarrow{md} \text{ end} \\
p & ::= \overrightarrow{d} e \quad \text{program}
\end{align*}

The metavariables \( m \) ranges over method names; \( f \) ranges over field names; \( x \) ranges over method parameter names; \( T \) ranges over trait names; \( O \) ranges over object names. We will use a right arrow overline notation to denote a possibly empty sequence (e.g., “\( \overrightarrow{a} \)” is a shorthand for a possibly empty sequence “\( a_1, \ldots, a_n \)”).

A program consists of a sequence of trait and object declarations and followed by a single top-level expression. Trait and object declarations consist of a header and a set of method declarations but headers of two types are slightly different. The header of trait declarations has a trait name to be defined, and a set of trait names to be extended. On the other hand, the header of object declarations has an object name to be defined, a list of fields that are passed in as parameters to the constructor, and a set of traits to be extended. Method declarations in a trait or an object may have the same name; FFMM allows method overloading.

FFMM has five expressions: parameter references, self references, constructor calls, field accesses, and method invocations. The keyword \texttt{self} is a special identifier that is like \texttt{this} in Java. FFMM has two kinds of types: trait types and object types. Trait types include the top trait \texttt{Object}, and object types are leaves of any FFMM type hierarchy, as can be seen in the syntax of FFMM.
FFMM has some internal symbols that do not appear in concrete syntax as follows:

\[
C ::= T \quad \text{trait or object name} \\
| O \quad \text{var} ::= x \quad \text{variable} \\
| \text{self} \\
\Gamma ::= \text{var} : \tau \quad \text{type environment}
\]

FFMM uses \(C\) to represent both a trait type \((T)\) and an object type \((O)\) simultaneously and treats parameter names and the \text{self} keyword as variables since the keyword \text{self} is a special form of a variable. Similar to other previous formal calculi, such as Featherweight Java (FJ) [19], FFMM has a type environment \(\Gamma\) and it is a set of pairs that consists of a variable name and a type.

Among other features of Fortress [10], FFMM does not include top-level variable and function definitions, functional method definitions (allow the \text{self} keyword as parameters of the method), excludes clauses, comprises clauses, where clauses, object expressions, and function expressions.

We make several simplifying assumptions about a program being type checked. These assumptions may be easily checked in a separate phase prior to the type checking phase and simplifying the typing rules of FFMM, as a lexical analysis phase simplifies the grammar and relieves the burden of a parser in ordinary compilers. Assumptions of FFMM are similar to assumptions of FJ and are as follows: (1) each name of a trait or an object is unique throughout the program, (2) every trait and object extends to at least one trait, (3) each name of the parents of a certain type is unique, (4) each name of a field is unique in the defining object, (5) no trait or object named \text{Object} is defined, (6) inheritance relation among traits does not make a cycle, and (7) each variable in a type environment \(\Gamma\) is unique.

3.3 Auxiliary Functions

We define auxiliary functions to look up methods to get visible methods from a given type. Two auxiliary functions to get visible methods are as follows:

\[
\text{defined}_p(\text{Object}) = \{\star\} \\
\text{defined}_p(C) = \{(md, C)\} \quad \text{where } C \rightarrow \text{md} \text{ end } \in p \\
\text{and visible}_p(C) = \text{defined}_p(C) \cup \bigcup_{T \in \{\Gamma\}} \text{visible}_p(T') \quad \text{where } C \rightarrow \text{extends } \{\overrightarrow{T}\} \rightarrow \in p
\]

The \text{defined}_p is a function that gathers defined methods in a given trait or a given object, and the \text{visible}_p function is a cumulative and recursive function that gathers all accessible methods from a given trait or a given object. A visible method set of a given type \(C\), written \text{visible}_p(C), returns a sequence of \{(md, C)\}, pairing a method declaration and a type name by which the method is defined for all methods declared in the class \(C\) and all of its super classes.

Two auxiliary functions in Figure 3.1 are related to the method selection. The \text{applicable}_p function determines whether each method in the \text{visible}_p set can be a candidate of a method call or not. It gets a method name, method argument types, and a method set as its inputs, and then returns an applicable method set. All methods in the applicable set have the same name as a given name and have super type argument of given argument types by the definition of the subtype relation (see Section 3.4). The \text{mostspecific}_p function selects an actual method to use in the method call. As the name indicates, it finds the most specific method in a given method set according to the
Most specific definitions:

\[
\text{mostspecific}_p((\{\text{md}, C\})) = \{\text{md}\}
\]

\[
\text{mostspecific}_p((\{\text{md}, C\})) = \begin{cases} 
\{\text{md}_i\} & \text{if}\ |\text{md}| = n \\
\text{md} = m((\cdots : \tau^d_1 : \tau^i_1 : \cdots m((\cdots : \tau^d_n) : \tau^r_n) \\
(\text{md}_i, C_i) \in \{(\text{md}, C)\}) \\
\forall 1 \leq j \leq n. (p \vdash \text{\tau}^a_j : \vdash \text{\tau}^a_j) \wedge p \vdash C_i <: C_j \\
\emptyset & \text{Otherwise}
\end{cases}
\]

Applicable definitions:

\[
\text{applicable}_p(m(\text{\tau}), \{(\text{md}, C)\}) = \{(\text{md}, C)\}
\]

\[
\text{applicable}_p(m(\text{\tau}), S) = \{(md, C) | (md, C) \in S, \text{md} = m(x : \text{\tau'}), p \vdash \text{\tau'} : \vdash \text{\tau}^a\}
\]

Figure 3.1: \text{applicable}_p and \text{mostspecific}_p functions

\[
\begin{array}{c}
\text{A} \\
\text{B} \quad \text{C} \\
\text{D} \\
\text{E}
\end{array}
\]

\[
\text{visible}_p(F) = \{m(A), m'(A), m(B), m'(B), m(C), m'(C), m(D), m(E), m'(E)\}
\]

Figure 3.2: A class hierarchy for examples of method selection

subtype relation. However, note that the \text{mostspecific}_p function only compares each parameter type of methods in a given set. If no unique most specific method exists in the given method set, the function returns an error (\emptyset) as its result. Therefore, the \text{mostspecific}_p function cannot guarantee that FFMM has no ambiguous method calls at run time by itself. However, FFMM always returns a single method as a result of the \text{mostspecific}_p function at run time if the program is well-typed because of the overloading rules in Figure 3.4. Figure 3.2 and the following examples show how the \text{applicable}_p function and the \text{mostspecific}_p function work.

- \text{applicable}_p(m(D), \text{visible}_p(F)) = \{m(A), m(B), m(C), m(D)\}
- \text{mostspecific}_p(\text{applicable}_p(m(D), \text{visible}_p(F))) = \{m(D)\}
- \text{applicable}_p(m'(D), \text{visible}_p(F)) = \{m'(A), m'(B), m'(C)\}
- \text{mostspecific}_p(\text{applicable}_p(m'(D), \text{visible}_p(F))) = \emptyset

3.4 Static Semantics

The subtype relation and the well-formed type relation of FFMM are similar to those of other formal calculi such as FJ [19]. In particular, the well-formed type relation,
implies exactly the same meaning as that of FJ. However, the subtype relation of FFMM slightly different with that of FJ since FFMM supports multiple inheritance. The subtype relation of FFMM is as follows:

\[ p \vdash \text{Object ok} \]

\[ \frac{- C \in p}{p \vdash C \text{ ok}} \]

Figure 3.3 shows an intersection type definition and the meet rule of FFMM. The meet of a set of types is the most specific type in the set, and an intersection type [18, Chapter 15.7] implies the greatest lower bound of two given types. For example, with the type hierarchy in Figure 3.2, \( B \cap C = D \) and \( D \cap E = E \), and with the type hierarchy in Figure 2.2, \( p \vdash \text{meet}(\{\text{Car}, \text{CampingTrailer}, \text{CampingCar}\}, \text{CampingCar}) \). An intersection type definition and the meet rule play a key role in ruling out ambiguities with symmetric multiple dispatch and symmetric multiple inheritance since we need to find a tie-breaking meet with the \( \text{collide} (\text{Vehicle v}, \text{Car c}) \) and the \( \text{collide} (\text{Car c}, \text{Vehicle v}) \) to solve the problem in Figure 2.3. Actually, the bug in CFWO that we found was in its definition of the meet rule.

The two typing rules in Figure 3.4 check all possible pairs of a set of visible methods to guarantee no ambiguous calls at run time in a well-typed program. Four sub rules in “valid declarations” rules exactly correspond to the overloading rules in Section 2.3: Exclusion Rule, Subtype Rule, and Meet Rule.

First, the \( \text{[VALIDEXC]} \) rule describes the Exclusion Rule, but the \( \text{[VALIDEXC]} \) is slightly weaker than the actual meaning of the Exclusion Rule that the Fortress team intended [9]. Fortress allows programmers to declare exclusive types using \( \text{excludes} \) clauses. For example, a trait or an object cannot extend a \( \text{String} \) type and a \( \text{Number} \) type simultaneously if the \( \text{String} \) type declares that the \( \text{Number} \) type is in the set of exclusive types of the \( \text{String} \) type using type exclusion relation. By guaranteeing the \( \text{String} \) type and the \( \text{Number} \) type are disjointed to each other, we can argue that the method \( m(\text{Number}) \) and the method \( m(\text{String}) \) cannot be applied at the same time. Therefore, we can determine that the method pair \( (m(\text{Number}), m(\text{String})) \) is a valid method pair. In contrast to the Fortress language and the Exclusion Rule in the previous work [9], we remove \( \text{excludes} \) clauses from FFMM since such clauses are largely orthogonal to multiple dispatch. Therefore, a pair of method declarations only satisfies the Exclusion Rule (\( \text{[VALIDEXC]} \) in FFMM) when two methods in the pair have different numbers of parameters. We leave adding \( \text{excludes} \) clauses in FFMM as our future work since such clauses can enhance the expressive power of programming languages when they are integrated with method overloading.

Second, the \( \text{[VALIDSUBTYR]} \) and \( \text{[VALIDSUBTYL]} \) rules correspond to the Subtype Rule. If the parameter type of one declaration is a strict subtype of the parameter of the other declaration and if the return type of the former is a subtype of the return type of the latter, then the pair satisfies the Subtype Rule. These rules guarantee that specialized method selection in run time is type safe. Therefore, by the \( \text{[VALIDSUBTYR]} \) and \( \text{[VALIDSUBTYL]} \) rules, the return type of the selected method at the run time for each method call should be a
Intersection type: \( \tau \cap \tau = \tau \)

\[
\tau_1 \cap \tau_2 = \begin{cases} 
\tau_3 & \text{if } \tau_3 <: \tau_1 \land \tau_3 <: \tau_2 \land \tau_1 \not<: \tau_2 \\
\tau_1 & \text{if } \tau_1 <: \tau_2 \\
\tau_2 & \text{if } \tau_2 <: \tau_1
\end{cases}
\]

Most specific type: \( p \vdash \text{meet}(\{\tau\}, \tau) \)

\[
\frac{\tau' \in \{\tau\} \quad p \vdash \tau' <: \tau' \quad p \vdash \{\tau\} <: \tau'}{p \vdash \text{meet}(\{\tau\}, \tau')} \quad [\text{MEET}]
\]

Figure 3.3: Intersection type and meet rule

subtype of the return type of the statically selected method that a type-checker already checked. On top of that, by the definitions of two rules ([VALIDSUBTYR] and [VALIDSUBTYL]), the return type for a specific method call can only be decreased. Therefore, these rules safely allow to call a more specialized method at run time than the selected method at static time.

Third, the [VALIDMEET] rule describes the Meet Rule. By the definition of the [VALIDMEET] rule, if two methods of a method pair are both applicable and they are not in subtype relation, a visible method set should have a method declaration whose each type of all parameters is the meet type with the types in the same position of parameters of two declarations and itself. This rule eliminates all possibilities of ambiguities in the method selection at run time by guaranteeing a hierarchy of parameter types of all methods in the visible method set as a meet semi-lattice structure with the type hierarchy of the program. For example, the \( \text{visible}_p(F) \) in Figure 3.2 should have \( m'(D) \) to satisfy the [VALIDMEET] rule.

As usual, a type judgement of the form \( p; \Gamma \vdash e : \tau \) states that “\( e \) has type \( \tau \) in the type environment \( \Gamma \)”.

While FFMM has five expressions, it has four expression typing rules since the [T-VAR] rule covers two expressions in FFMM (parameter references and self references). Finally, the [T-TRAITDEF] and [T-OBJECTDEF] rules guarantee that any overloaded methods defined in the class are in a valid method set according to the rules discussed in Figure 3.4.
Valid method declarations:

\[ p \vdash \text{validMeth}(C) \]

\[ \forall \{(md, C), (md', C')\} \subseteq \text{visible}_p(C^o). \]

\[ md \neq md', \text{ (not same declaration)} \]

\[ md = m(\_ : \tau^o \rightarrow \_), \]

\[ md' = m(\_ : \tau'^o \rightarrow \_), \]

\[ p \vdash \text{valid}(m, C, \tau^a \rightarrow \tau'^a, C'^o, \text{visible}_p(C^o)) \]

\[ p \vdash \text{validMeth}(C^o) \]

Valid declarations:

\[ p \vdash \text{valid}(m, C, \tau^a \rightarrow \tau, C', \tau^a \rightarrow \tau', \{(md, \tau^a)\}) \]

\[ \left[ \text{VALIDEXC} \right] \]

\[ |\tau^a| \neq |\tau'^a| \]

\[ p \vdash \text{valid}(m, C, \tau^a \rightarrow \tau'^a, C', \tau'^a \rightarrow \tau'^', S) \]

\[ |\tau^a| = |\tau'^a| \quad C \tau^a \neq C' \tau'^a \quad p \vdash \tau'^a <: \tau'^a \]

\[ p \vdash \tau'^a <: \tau' \quad p \vdash C' <: C \]

\[ \left[ \text{VALIDSUBTYR} \right] \]

\[ |\tau^a| = |\tau'^a| \quad C \tau^a \neq C' \tau'^a \quad p \vdash \tau'^a <: \tau'^a \]

\[ p \vdash \tau'^a <: \tau' \quad p \vdash C' <: C' \]

\[ p \vdash \text{valid}(m, C, \tau^a \rightarrow \tau', C', \tau'^a \rightarrow \tau'^', S) \]

\[ \left[ \text{VALIDSUBTYL} \right] \]

\[ l = |\tau^a| = |\tau'^a| \quad C \tau^a \neq C' \tau'^a \quad \tau_0^a = C \quad \tau_0'^a = C' \]

\[ \exists (m(\_ : \tau^o \rightarrow \_), \_ : \tau^o_0 \rightarrow \_)) \in S. \]

\[ \left[ \text{VALIDMEET} \right] \]

\[ (l = |\tau^a|) \land (\forall 0 \leq i \leq l, p \vdash \text{meet}(\{\tau_i^a, \tau_i'^a, \tau_i'^o\}, \{\tau_i'^{o'}\})) \]

\[ p \vdash \text{valid}(m, C, \tau^a \rightarrow \tau', C', \tau'^a \rightarrow \tau'^', S) \]

Figure 3.4: Overloading rules in FFMM
Expression typing: $p; \Gamma \vdash e : \tau$

[T-VAR] \( \frac{\text{var} \in \text{dom}(\Gamma)}{p; \Gamma \vdash \text{var} : \Gamma(\text{var})} \)

[T-OBJECT] \( \frac{\text{object } O(f : \tau) \text{ end} \in p}{p; \Gamma \vdash O(f) : \tau} \)

[T-FIELD] \( \frac{p; \Gamma \vdash e_o : O \text{ object } O(f : \tau) \text{ end} \in p}{p; \Gamma \vdash e_o.f : \tau} \)

[T-METHOD] \( \frac{\text{mostspecific}_p(\text{applicable}_p(m(f, \tau), \text{visible}_p(\tau_o))) = \{m(- : \tau)\}}{p; \Gamma \vdash e_o.m(f) : \tau'} \)

Figure 3.5: Expression typing rules

Program typing: $\vdash p : \tau$

[T-PROGRAM] \( \frac{p = \overrightarrow{d} e \quad p \vdash \overrightarrow{d} \text{ ok} \quad p; \emptyset \vdash e : \tau}{\vdash p : \tau} \)

Trait or object definition typing: $p \vdash \overrightarrow{d} \text{ ok}$

[T-TraitDef] \( \frac{p \vdash \overrightarrow{T} \text{ ok} \quad p; \text{self} : T \vdash \overrightarrow{md} \text{ ok} \quad p \vdash \text{validMeth}(T)}{p \vdash \text{trait } T \text{ extends } \{T\} \text{ md end ok}} \)

[T-ObjectDef] \( \frac{p \vdash \overrightarrow{T} \text{ ok} \quad p \vdash \overrightarrow{md} \text{ ok} \quad p; \text{self} : O f : \tau_1 \vdash \overrightarrow{md} \text{ ok} \quad p \vdash \text{validMeth}(O)}{p \vdash \text{object } O(f : \tau_1) \text{ extends } \{T\} \text{ md end ok}} \)

Method typing: $p; \Gamma \vdash \overrightarrow{md} \text{ ok}$

[T-MethodDef] \( \frac{p \vdash \overrightarrow{md} \text{ ok} \quad p \vdash \tau_0 \text{ ok} \quad p; \Gamma x : \overrightarrow{\tau} \vdash e : \tau' \quad p \vdash \tau' <: \tau_0}{p; \Gamma \vdash m(x : \overrightarrow{\tau}) : \tau_0 = e \text{ ok}} \)

Figure 3.6: Definition typing rules
Value, evaluation contexts and redexes

\[ v ::= O(\overline{v}) \] \hspace{1cm} \text{value} \\
\[ E ::= \Box \] \hspace{1cm} \text{evaluation context} \\
\[ \mid O(\overline{v} E \overline{v}') \] \\
\[ \mid E.f \] \\
\[ \mid e.m(\overline{v}) \] \\
\[ \mid e.m(\overline{v} E \overline{v}') \] \\
\[ R ::= v.f \] \hspace{1cm} \text{redex} \\
\[ \mid v.m(\overline{v}) \]

Evaluation rules: \[ p \vdash E[R] \rightarrow E[e] \]

[R-FIELD] 
\[
\begin{align*}
\text{object } O - (f: & \overline{\_}) - \text{end} \in p \\
\frac{p \vdash E[O(\overline{v}).f_i] \rightarrow E[v_i]}{E[O(\overline{v}).f_i]}
\end{align*}
\]

[R-METHOD] 
\[
\begin{align*}
\text{object } O - \text{end} \in p \\
\frac{\text{mostspecific}_p(\text{applicable}_p(m(\overline{v})), \text{visible}_p(O))) = \{m(\overline{x}: \overline{\_}) : \overline{\_} = e\}}{p \vdash E[O(\overline{v}).m(\overline{v}')] \rightarrow E[[O(\overline{v}).\text{self}]][\overline{v}' / \overline{v}][e]}
\end{align*}
\]

Type of values: \[ \text{type}(v) = \tau \]

\[ \text{type}(O(\overline{v})) = O \]

Figure 3.7: Dynamic semantics

3.5 Dynamic Semantics

The dynamic semantics of FFMM is provided in Figure 3.7. It presents definitions of a value, evaluation contexts, and redex as well as evaluation rules. The FFMM dynamic semantics consist of two evaluation rules: one for field accesses and another for method invocations. We assume that the FFMM dynamic semantics does not expand across definition boundaries unless the entire definition is included in it.
Chapter 4. FFMM in CoQ

4.1 Assumptions of FFMM in CoQ

Before representing FFMM syntax and semantics, we need to find a proper way to express several assumptions of FFMM described in Section 3.2: (1) each name of a trait or an object is unique throughout the program, (2) every trait and object extends to at least one trait, (3) each name of the parents of a certain type is unique, (4) each name of a field is unique in the defining object, (5) no trait or object named Object is defined, (6) inheritance relation among traits does not make a cycle, and (7) each variable in a type environment \( \Gamma \) is unique.

The existing metatheory library of Cast-free Featherweight Java (CFFJ) [24] has definitions to cover three of those assumptions: (1), (4), and (7). We can define most declarations in FFMM, a trait table, an object table (see Section 4.2), field declarations, method declarations, and \( \Gamma \), using a list of pairs that have an identifier in their first position and contents of each identifier in their second position. For example, field declarations in one object are represented as a list of \((f, \tau)\) pairs, every \(f\) is an identifier for each pair, and every \(\tau\) means contents for each corresponding \(f\). Therefore, if we guarantee that each identifier appears at most once in the first position of all elements in the list, we can represent the assumptions, (1), (4), and (7), properly. In the metatheory library of CFFJ, the inductively defined proposition \(\text{ok}\) is as follows:

\[
\text{Inductive } \text{ok} : \text{list } (\text{atom} \times A) \rightarrow \text{Prop} := \\
\begin{align*}
\text{| ok.nil: } & \text{ok nil} \\
\text{| ok.cons: } & \forall E x v, \text{ok } E \rightarrow \text{no.binds } x E \rightarrow \text{ok } ( (x, v) :: E).
\end{align*}
\]

guaranteeing those properties. We can simply check the assumptions, (1), (2), and (7), by checking a trait table, an object table, field declarations, and \(\Gamma\) satisfies all conditions of the \(\text{ok}\) definition.

To express two assumptions of FFMM, (2) and (3), we extend the existing metatheory library of CFFJ by adding the \(\text{ok.list}\) definition. The \(\text{ok.list}\) definition is as follows:

\[
\text{Inductive } \text{ok.list} : \text{list } \text{atom} \rightarrow \text{Prop} := \\
\begin{align*}
\text{| ok.one: } & \forall a, \text{ok.list } (a :: \text{nil}) \\
\text{| ok.cons': } & \forall a al, \text{ok.list } al \rightarrow a \not\in \text{al} \rightarrow \text{ok.list } (a :: al).
\end{align*}
\]

guaranteeing that the list is not \(\text{nil}\) and every element is unique in the list. Therefore, using the \(\text{ok.list}\) definition, we can simply represent assumptions, (2) and (3), that cannot be covered by the \(\text{ok}\) definition.

As we discussed, the metatheory library can cover most of assumptions for FFMM. However, two assumptions, (5) and (6), are not covered by the metatheory library. On top of that, when we prove FFMM type safety, we have to use the assumption, “The program is well-typed.” One way to add those assumptions in the proof process is to make those assumptions as hypotheses or axioms in the global scope using the Hypothesis or Axiom keyword in CoQ. However, to separate the concerns of those assumptions, we use a module system [27, Chapter5] in CoQ. Figure 4.1 shows the module in which three assumptions are defined and how to use those assumptions. In the Hyps module, the \(\text{tt.noobj}\) corresponds to the assumption (5), and the \(\text{Prog.ty}\) and the \(\text{Acyclic}\) correspond to the assumption, “The program is well-typed,” and assumption (6), respectively.

– 14 –
Module Type HYPs.
 Parameter \texttt{tl\_noobj} : Object \notin \text{dom} TT.
 Parameter \texttt{ok\_prog} : Prog\_ty.
 Parameter \texttt{acyclic} : Acyclic.
 End HYPs.

\ldots
Module PROPERTIES (H: HYPs).
\ldots

Figure 4.1: HYPs module in COQ

\section{Syntax and Semantics of FFMM in COQ}

While the COQ implementation of FFMM is very similar to the FFMM calculus, some parts are different in the implementation. We modify some definitions or rules in FFMM to easily implement FFMM using COQ and mechanize FFMM type safety proof. Among those modifications in COQ, this section illustrates some important changes: a program definition, method declarations, and the ”valid method declarations” rule and ”valid declarations” rules. For other changes, we omit explanation, since they are easily recognizable, and explaining those changes are too tedious.

\textit{Separate a class table into two class tables: an object table and a trait table}

Java-like languages provide only classes, but FFMM provides both traits and objects, as we discussed in Section 3.2. In FFMM calculus, we use a metavariable \( C \) to represent both a trait and an object at the same time. Therefore, we can represent an FFMM program as a tuple of class table, \( CT \), and one top-level expression, \( e \), in the same way as FJ to represent a program. However, our COQ implementation maintains two separate class tables: an object table, \( OT \), and a trait table, \( TT \). Therefore, the program definition is also changed as a tuple of \((OT, TT, e)\). By separating two class tables, we can reuse the \texttt{Metatheory} library of CFFJ instead of making new definitions to represent FFMM.

\textit{Add a method identifier to each method}

In FFMM, each method is identified by its name and parameter types. However, in our implementation, we add a method identifier that has an \texttt{atom} type (see Section 4.3) for each method to allow overloaded methods and to identify each method in a set of overloaded methods. In particular, the method definition in FFMM syntax (see Section 3.2) was \( md::= m(\vec{x}:\tau) : \tau = e \). However, in our implementation, we change this definition as follows: \texttt{Notation mth := (mid \times (mname \times env \times typ \times exp))\%type}. By adding a method identifier to each method, we can allow method overloading and identify each method in a set of overloaded methods without adding additional definitions in the \texttt{Metatheory} library.

\textit{Modify the overloading rules of FFMM}

We modify the [VALIDMETH] rule and ”valid declarations” rules in Figure 3.4 slightly. Figure 4.2 shows how ”valid declarations” rules are defined in our COQ implementation. Major modifications are two parts.

First, we remove all preprocessing in the the [VALIDMETH] rule. By the definition of the [VALIDMETH] rule, FFMM does not check the ”valid declarations” rules for a pair of method declarations if two methods in the pair are exactly the same methods or two methods in the pair have different names. However, in our implementation,
Definition validmeet' (mn: mname) (ty: list typ) (ty: typ) (mS: mSet): Prop :=
exists2 mdt, mdt \in mS \&
((ty = (getartys mdt)) \& (ty = (snd mdt)) \& (mn = (getmname mdt)))

Inductive valid (mdt1 mdt2 : mdtype) (mS: mSet): Prop :=
| valid_same : 
  (getmid mdt1) = (getmid mdt2) \Rightarrow (snd mdt1) = (snd mdt2) \Rightarrow
  valid mdt1 mdt2 mS
| valid_diff_name : 
  (getmname mdt1) \neq (getmname mdt2) \Rightarrow
  valid mdt1 mdt2 mS
| valid_exc : 
  (getmid mdt1) \neq (getmid mdt2) \lor (snd mdt1) \neq (snd mdt2) \Rightarrow
  (getmname mdt1) = (getmname mdt2) \Rightarrow (getenvlen mdt1) \neq (getenvlen mdt2) \Rightarrow
  valid mdt1 mdt2 mS
| valid_sub_ty_r : 
  (getmid mdt1) \neq (getmid mdt2) \lor (snd mdt1) \neq (snd mdt2) \Rightarrow
  (getartys mdt1) \neq (getartys mdt2) \lor (snd mdt1) \neq (snd mdt2) \Rightarrow
  (getmname mdt1) = (getmname mdt2) \Rightarrow (getenvlen mdt1) = (getenvlen mdt2) \Rightarrow
  sub_tys (getartys mdt2) (getartys mdt1) \Rightarrow
  sub_ty (getrty mdt2) (getrty mdt1) \Rightarrow
  sub_ty (snd mdt2) (snd mdt1) \Rightarrow
  valid mdt1 mdt2 mS
| valid_sub_ty_l : 
  (getmid mdt1) \neq (getmid mdt2) \lor (snd mdt1) \neq (snd mdt2) \Rightarrow
  (getartys mdt1) \neq (getartys mdt2) \lor (snd mdt1) \neq (snd mdt2) \Rightarrow
  (getmname mdt1) = (getmname mdt2) \Rightarrow (getenvlen mdt1) = (getenvlen mdt2) \Rightarrow
  sub_tys (getartys mdt1) (getartys mdt2) \Rightarrow
  sub_ty (getrty mdt1) (getrty mdt2) \Rightarrow
  sub_ty (snd mdt1) (snd mdt2) \Rightarrow
  valid mdt1 mdt2 mS
| valid_meet : \forall tys ty,
  (getmid mdt1) \neq (getmid mdt2) \lor (snd mdt1) \neq (snd mdt2) \Rightarrow
  (getartys mdt1) \neq (getartys mdt2) \lor (snd mdt1) \neq (snd mdt2) \Rightarrow
  (getmname mdt1) = (getmname mdt2) \Rightarrow (getenvlen mdt1) = (getenvlen mdt2) \Rightarrow
  \neg (sub_tys (getartys mdt1) (getartys mdt2)) \Rightarrow
  \neg (sub_tys (getartys mdt2) (getartys mdt1)) \Rightarrow
  is_tys (getartys mdt1) (getartys mdt2) tys \Rightarrow
  is_ty (snd mdt1) (snd mdt2) ty \Rightarrow
  validmeet' (getmname mdt1) tys ty mS \Rightarrow
  valid mdt1 mdt2 mS.
we do not include those preliminary inspections before sending a method pair to “valid declaration” rules to check for simplicity. Instead of those checks in the \[\text{VALIDMETH}\] rule, we add two more constructors in “valid declarations” rules, and those two constructors replace the preprocessing of the \[\text{VALIDMETH}\] rule properly. First, one is the valid_same constructor, and if we check the overloading rules with a pair of methods in which two elements of the pair are exactly same, this constructor is directly satisfied. The other is the valid_diff_name constructor, and if we check the overloading rules with a pair of method declarations that have different names, this constructor is satisfied vacuously.

The second modification in the “valid declarations” rules (the overloading rules) is related to the \[\text{VALIDMEET}\] rule in Figure 3.4. In fact, the overloading rules in FFMM are not deterministic. In particular, the \[\text{VALIDMEET}\] rule in the FFMM calculus could be satisfied by a pair of method declarations whose parameter types of the method are in the subtype relation, even thought the \[\text{VALIDSUBTYL}\] or \[\text{VALIDSUBTYR}\] rule also checks the pair of method declarations repeatedly. On the contrary, every constructor of the valid definition in our implementation is disjointed. The valid_meet constructor is disjointed to the valid_sub_ty_r and valid_sub_ty_l constructors since we add \(\text{sub_tys (getartys mdt1) (getartys mdt2)}\) and \(\sim (\text{sub_tys (getartys mdt2) (getartys mdt1)})\) conditions in the valid_meet constructor. By making the overloading rules in COQ implementation deterministic, we can reduce some cases for the proofs of the properties that are related to the overloading rules in FFMM type safety proof.

Because the differences in the calculus and the implementation are minor implementation details, we believe that our implementation faithfully and correctly reflects the syntax and semantics of FFMM.

### 4.3 FFMM in COQ Summary

Table 4.1 briefly summarizes the details of COQ libraries that is related to the syntax and semantics of FFMM. Among three libraries, the Atom library has a definition of an atom type that are used to represent every metavariable such as \(T\), \(O\), and \(m\) in our implementation. We brought this library from the COQ implementation of CFFJ [24], and the actual type of the atom type is a nat type that is defined in the COQ standard library [28]. As we discussed in Section 4.1, the Metatheory library has several definitions to represent the assumptions of FFMM. On top of that, this library has several other definitions that help to implement FFMM calculus using COQ and some proofs that are related to the definitions defined in the Metatheory library. The FFMM_Definitions library represents the syntax and semantics of FFMM, and it also has the HYPS module in Figure 4.1. Whole libraries are available online [17].

<table>
<thead>
<tr>
<th>Library Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atom</td>
<td>Definition of an atom type</td>
</tr>
<tr>
<td>Metatheory</td>
<td>Definitions to represent object oriented languages using COQ</td>
</tr>
<tr>
<td>FFMM_Definitions</td>
<td>Main file to describe the syntax and semantics of FFMM</td>
</tr>
</tbody>
</table>

Table 4.1: COQ libraries to define FFMM calculus
Chapter 5. Type Safety Proof in COQ

5.1 Properties

We use one library in CFFJ and implement seven libraries to prove FFMM type safety. Every library that we made has the same structure: First, the library states proofs of some properties that do not need the assumption, “The program is well-typed,” and then, proofs of the properties that need the assumptions are as follows. Among approximately 150 facts, lemmas, and theorems in our type safety proof, we will state important properties that are closely related to multiple dispatch, multiple inheritance and type safety.

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

**Lemma 5.1.1.** Suppose that \( p \) is well-typed. If \( \text{applicable}_p(m(x \rightarrow \tau), \text{visible}_p(C)) \) = \( \{m, \mathring{\tau}, C\} \) and \( \{m, \mathring{\tau}, C\} \neq \emptyset \), then there exist \( m'd \) such that \( \text{mostspecific}_p(m, \mathring{\tau}, \Gamma) \) = \{\( m'd \)\}.

This lemma implies that there are no ambiguous method calls at run time in FFMM. A \( \{m, \mathring{\tau}, C\} \neq \emptyset \) assumption in the lemma 5.1.1 is related to the assumptions of well-typed programs in Section 3.2. In a well-typed program, each method call has at least one applicable method by the \([\text{T-METH} \rightarrow \text{METHOD}]\) rule in Figure 3.5. It also implies that there are no undefined method calls at run time in FFMM. This lemma plays an important role in the proof of the following two lemmas;

**Lemma 5.1.2.** Let \( p \) be well-typed, if \( \text{mostspecific}_p(\{\text{applicable}_p(m(x \rightarrow \tau), \text{visible}_p(C))\}) = \{m(x \rightarrow \tau': \tau = e)\} \) and \( p \vdash \Gamma \vdash \tau' \vdash \Gamma \), then there exists \( m(x \rightarrow \tau') : \tau = e \) such that \( \text{mostspecific}_p(\{\text{applicable}_p(m(x \rightarrow \tau'), \text{visible}_p(C))\}) = \{m(x \rightarrow \tau') : \tau = e\} \) and \( p \vdash \tau' \vdash \tau'' \).

**Lemma 5.1.3.** Let \( p \) be well-typed, if \( \text{mostspecific}_p(\{\text{applicable}_p(m(x \rightarrow \tau), \text{visible}_p(C'))\}) = \{m(x \rightarrow \tau': \tau = e)\} \) and \( p \vdash \Gamma \vdash C \vdash \Gamma \), then there exists \( m(x \rightarrow \tau') : \tau = e \) such that \( \text{mostspecific}_p(\{\text{applicable}_p(m(x \rightarrow \tau'), \text{visible}_p(C))\}) = \{m(x \rightarrow \tau') : \tau = e\} \) and \( p \vdash \tau' \vdash \tau'' \).

Lemmas 5.1.2 and 5.1.3 show that choosing a more specific method at run time than the statically chosen method is type safe. Both lemmas are related to the \([\text{VALIDSUBTYR}]\) and \([\text{VALIDSUBTYL}]\) rules in Figure 3.4. With multiple dispatch, the method to be called at run time might be different from the method that is selected in the static time, which the type-checker checks whether the program is well-typed or not. However, by those two lemmas, the return type in the method selection at run time always satisfies type safe. These lemmas are important for the proof of term substitution preserve typing (lemma 5.1.4) and the preservation theorem (theorem 5.1).

Term substitution preserve typing is as follows;

**Lemma 5.1.4** (Term substitution preserve typing). Suppose that \( p \) is well-typed. If \( p; \Gamma \vdash x : \tau \) and \( p; \Gamma \vdash e : \tau \) and \( p \vdash \tau' \vdash \gamma', \) then \( p; \Gamma \vdash [\gamma'/x]e : \tau' \) for some \( \tau' \) such that \( p \vdash \tau' \vdash \gamma' \).

Then, proving the following two theorems is an usual and traditional step to show the type safety of a calculus;

**Theorem 5.1.1** (Preservation). Suppose that \( p \) is well-typed. If \( p; \Gamma \vdash e : \tau \) and \( p \vdash e \rightarrow e' \), then \( p; \Gamma \vdash e' : \tau' \) where \( p \vdash \tau' \vdash \gamma' \).

This theorem is proved by the induction on a derivation of \( p \vdash e \rightarrow e' \) rules. The proof of the preservation theorem using COQ is given in Figure 5.1
Theorem preservation: $\forall E \; e \; e' \; t. \;
\text{exp}\_\text{ty} \; E \; e \rightarrow \text{eval}\_\text{rule} \; e \; e' \rightarrow \text{wide}\_\text{typing} \; E \; e' \; t.$

Proof.

intros $E \; e \; e' \; t \; H \; H0.$ generalize dependent $t.$
induction $H0.$ intros.

Case "eval\_field".

inversion $H2.$ inversion $H6.$ subst.
destruct $H8$ as ($fs1$, $H8a$, $H8b$).
assert ($fs0 = fs1$) by (eapply ob\_fields\_fun; eassumption); subst.
assert ($fs1 = fs$) by (eapply ob\_fields\_fun; eassumption); subst.
destruct binds\_zip with (1:=H15) (2:=H0) (3:=H8b) as ($e0$, $Hb1$, $Hb2$).
assert ($fs0 = fs1$) by eauto using mostspecific\_implies\_sub\_ty.
assert ($sub\_ty\_t\; t\; t0$) by eauto using sub\_argtys\_mostspecific\_implies\_sub\_ty.
subst injections.
destruct method\_implies\_typing with (1:=H0) (2:=H1) (3:=H2) as ($t'$, $H9a$, $H9b$).
eapply term\_substitutivity; (try simpl; eauto).
assert ($sub\_ty\; t\; (\text{imgs} \; E0))$ by eauto using mostspecific\_implies\_super\_ty.
assert ($\text{wide}\_\text{ typings} \; E \; e \; t\; t0$) by eauto using val\_tys\_and\_wide\_typings\_implies\_exp\_tys.

Case "eval\_context".

eapply preservation\_over\_ec; try eauto.
Qed.

Figure 5.1: Preservation theorem proof in COQ

Theorem 5.1.2 (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 $p \vdash e \rightarrow e'$.

This theorem is proved by the induction on a derivation of $p; \Gamma \vdash e : \tau$ rules. In the case of the \[T-METHOD\] rule, we need to find a witness ($e'$) if all arguments and the method owner are values. Therefore, the lemma 5.1.1 plays an important role in the proof of the progress theorem. The proof of the progress theorem using COQ is given in Figure 5.2.

By theorems 5.1.2 and 5.1.1, we can get the following theorem directly:

Theorem 5.1.3 (Type Safety). Suppose that $p$ is well-typed. If $p; \emptyset \vdash e : \tau$ and $p \vdash e \rightarrow ^* v$, then $p; \emptyset \vdash v : \tau'$ and $p \vdash \tau' <: \tau$. 
Theorem progress':
∀ E e t. exp Ty E e t → E = nil → val e ∨ ∃ e', eval_rule e e' ∧
∀ E e t. wide_typing E e t → E = nil → val e ∨ ∃ e', eval_rule e e' ∧
∀ E env E env → E = nil → (∀ v, ln v ds → val v) ∨
exists2 EE, exps_context EE & (exists2 e0, (EE e0) = ds & (∃ e0', eval_rule e0 e0'))).

Proof.
apply typings_mutind; intros; subst E; specialize trivial; try (contradiction (binds_nil H0); fail).
Case "lernew".
destruct H2 as [H2 | (EE,H2a,(e0,H2b,(e0',H2c)))] [auto | ]; subst ex; eauto.
Case "lfield".
destruct H0 as [H0 | (e',H0)]; [ right; eauto ].
destruct H0; inversion H; subst.
destruct wide_typings_imply.zip with (1 := H8) as (Eds, Hzip).
destruct H1 as (f0,H1a,H1b).
assert (f0 = fs) by eapply ob_fields; assumption; subst.
destruct binds.zip with (1 := H8) (2 := Hzip) (3 := H1b); eauto.
Case "lmeth".
destruct H0 as [H0 | (e',H0)]; [ right; eauto ].
destruct H0; inversion H; subst. destruct H2.
SCase "forall v. exp v "in ex v" ; val v".
assert (∃ tys, val tys ex tys) by eauto using vals_always_has_val tys.
destruct H6 as (tys', H6).
assert (sub_tys tys' ts) by eauto using val_tys_and-wide_typings_imply_sub_tys.
assert (exists2 AmS', applicable mn tys' ms AmS' & (∀ mult, mult \in AmS → mult \in AmS'))
  by eauto using argty_imply_mostspecific_sub_set.
destroy H8 as (AmS', H8a, H8b).
assert (∃ mi, ∃ e, ∃ E, exists2 rty', mostspecific (mi, (mn, E, rty', e)) AmS' & sub_ty rty' t)
  by eauto using sub_argtys_has_mostspecific.
destruct H8 as (mi', (e', (E', rty', H8a, H8b))).
assert (sub_tys tys' (imgs E')) by eauto using mostspecific_imply_mostspecific_super_ty.
assert (wide_typings nil es tys') by eauto using val_tys_and-wide_typings_imply_exp_tys.
assert (wide_typings nil es (imgs E')) by eauto.
destruct wide_typings_imply.zip with (1 := H12) as (Eds, Hzip).
right; (subt_exp (self, e, new on es0) :: Eds) e); eauto.
SCase "exists2 EE : exp q; list exp .".
destruct H2 as (EE, H2a, (e0, H2b, (e0', H2c)));
subst.
right; (e_method (e, new on es0) mn (EE e0')) ; eauto.
Case "wsnil". left; intros; contradiction H0.
Case "wsccons".
destruct H0 as [H0 | (EE, H0a, (e0, H0b, H0c))].
SCase "values es". destruct H2 as [H2 | (e', H2)].
SCase "valse'".
  left; intros.
destruct ln_inv with (1 := H3); [ subst ]; eauto.
SCase "e_progres". right; (fun e1 ⇒ e1 := e); eauto.
SCase "es_progres". subst ex; right; (fun e1 ⇒ e := (EE e1)); eauto.
Qed.

Theorem progress. ∀ e t,
exp Ty nil e t → val e ∨ (∃ e', eval_rule e e').

Proof.
intros. destruct (progress') as (Hpro_); apply Hpro with (1 := H); reflexivity.
Qed.

Figure 5.2: Progress theorem proof in CoQ
<table>
<thead>
<tr>
<th>Library Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdditionalTactics</td>
<td>Additional tactic definitions</td>
</tr>
<tr>
<td>OkCTableFacts</td>
<td>Properties related to the <code>tt_noobj</code> and <code>Prog_ty</code> assumptions (see Figure 4.1)</td>
</tr>
<tr>
<td>UniquenessFacts</td>
<td>Properties related to the uniqueness</td>
</tr>
<tr>
<td>SubTypeFacts</td>
<td>Properties related to the type definition and the subtype relation</td>
</tr>
<tr>
<td>VisibleFacts</td>
<td>Properties related to the visible auxiliary function</td>
</tr>
<tr>
<td>ApplicableFacts</td>
<td>Properties related to the applicable auxiliary function</td>
</tr>
<tr>
<td>TypingFacts</td>
<td>Properties related to expression typing rules</td>
</tr>
<tr>
<td>TypeSafety</td>
<td>Main file for type safety proof</td>
</tr>
</tbody>
</table>

Table 5.1: COQ libraries to prove FFMM type safety

5.2 Proof Summary

Table 5.1 briefly summarizes details of COQ libraries that are related to FFMM type safety proof. Among eight libraries, the AdditionalTactics library [29] has definitions of additional tactics that help several proofs for our facts, lemmas, and theorems. The OkCTableFacts library consists of some proofs that are directly related to two assumptions: `tt_noobj` and `Prog_ty`. In the UniquenessFacts library, we prove the uniqueness properties of FFMM such as some facts like “If the method owner type and the method identifier are the same, then a corresponding method declaration is unique,” and the SubtypeFacts library has proofs that are related to the type definition and the subtype relation of FFMM. One of the important facts in the SubtypeFacts library is the decidability of the subtype relation. The VisibleFacts, the ApplicableFacts, and the TypingFacts libraries consist of several proofs related to the `visible_p` function, the `applicable_p` function, and expression type rules, respectively. Those three libraries contain some important proofs for FFMM type safety. For example, lemma 5.1.1 is in the ApplicableFacts library and lemmas 5.1.2 and 5.1.3 are in the TypingFacts library. Finally, the TepeSafety library contains the most important lemmas and theorems for the type safety proof of FFMM. It has a weakening lemma, lemma 5.1.4, theorem 5.1.1, theorem 5.1.2, and theorem 5.1.3. The full proof of all the facts, lemmas, and theorems in our implementation is available online [17], as are the libraries that are related to the calculus definition.
Chapter 6. Comparing Existing Coq Implementations

6.1 CFFJ, FBCF, and FFMM

Before FFMM, we mechanized Featherweight Basic Core Fortress (FBCF) [26], a very small core of the Fortress programming language, using Coq. On top of that, we found some object oriented programming languages that are mechanized by Coq. Among those Coq implementations, CFFJ, FBCF, and FFMM share similar Metatheory libraries. CFFJ is quite similar to FJ [19], but it does not have casting expressions. FBCF has a trait type and an object type like FFMM. However, unlike FFMM and similar to CFFJ, FBCF provides method overriding and single inheritance. Therefore, FBCF uses similar method to CFFJ for method lookup and uses the \(mtype_p\) and \(mbody_p\) auxiliary functions. These functions traverse up the type hierarchy one by one until the functions find the intended method. On the contrary, FFMM uses the \(visible_p\) function for method lookup as we discussed in Section 3.3. Therefore, we can first traverse up the type hierarchy to gather all visible methods.

We argue that adding multiple inheritance to FFMM is much easier and more natural than adding it to FBCF when we mechanize both calculus using Coq. By traversing up the type hierarchy, we can simply add multiple inheritance using the `with` keywords in Coq that are used to represent mutually inductively defined proposition [20, Chapter 14.3]. However, if we want to add multiple inheritance with method lookup functions that traverse up the type hierarchy one by one, several additional definitions have to be added for the desired functionality.

Table 6.1 shows brief comparison of CFFJ, FBCF, and FFMM. In the table, each number corresponds to the number of lines of Coq implementations. The number in the specification column means how many lines are used for definitions, functions, and statements of facts, lemmas and theorem. The number in the proofs column means that how many lines are used to prove the facts, lemmas and theorems. We count those lines using a `coqwc` command that Coq provides.

The metatheory column contains the information of two libraries: Atom and Metatheory (see Section 4.3). Since we used exactly the same Metatheory library as CFFJ to mechanize FBCF, the values in the metatheory column for FBCF and CFFJ are exactly the same. On the contrary, as we discussed in Section 4.1, we define an additional definition, \(ok\_list\), in the Metatheory library of our Coq implementation. On top of that, for FFMM type safety proof, we also add several facts and lemmas that are related to the \(ok\) definition in the Metatheory library and the list definition in the Coq standard library. Therefore, the number in the metatheory column of FFMM is slightly larger than those of CFFJ and FBCF.

The number in the calculus column implies the number of lines that are used to represent the syntax and semantics of the language, and the number in the type safety proof column shows how many lines are used to state facts, lemmas, and theorems that we have to prove, and are used to prove those properties. When we take a look at each value in the total column for CFFJ and FBCF, values are quite similar even though the two calculi are slightly different. We used the same approach as FFMM (see Section 4.2) to represent different class declarations of FBCF, so when compared to CFFJ, additional definitions need to represent the syntax and semantics of FBCF for the method lookup functions and definition typing rules. Therefore, the number of lines for calculus definition of FBCF is slightly larger than that of CFFJ. However, the number of lines for specifications and proofs for the type safety proof of FBCF is almost the same as that of CFFJ. We think that the reason of these similarity is due to the difference of field declarations and class declarations. Compared to CFFJ, FBCF needs
additional lemmas and facts for its type safety proof when properties of the language are related to both types, but we can omit some cases that are related to field declarations since only object types can have field declarations in FBCF unlike CFFJ. Therefore, the number of lines in the total column of CFFJ is similar to that of FBCF. However, the whole size of the Coq code for FFMM is about three times bigger than that of FBCF. We think that crucial reasons of additional size are as follows: (1) changes in auxiliary functions related to method lookup and method selection, (2) changes in expression typing rules and evaluation rules that are related to method invocations, and (3) additional definitions related to the static overloading rules. We check our assumptions by looking through our calculus definitions in the FFMM_Definitions library and all libraries for FFMM type safety proof, and conclude that most of additional lines are due to adding multiple dispatch. On the contrary, burdens for multiple inheritance are not significant. Therefore, we can guess that we use about 1900 lines to state definitions and prove properties that are related to multiple dispatch.

### 6.2 CFFJ, LJ, and FGJ_Ω

CFFJ, Lightweight Feature Java (LJ) [21], and Featherweight Generic Java _Ω_ (FGJ_Ω) [11, 24] are different subset languages of Java. While CFFJ has four expressions (variables, field accesses, method invocations, and class creations), LJ has seven statements (blocks, variable assignments, field reads, field writes, conditional branches, object creations, and method calls). FGJ_Ω_ is an extension of FGJ with variables representing type constructors, and has four expressions: variables, field accesses, method invocations, and class creations, just as like CFFJ. It is hard to compare those Coq implementations precisely since each Coq implementation uses a different style to mechanize its type safety proof. However, by comparing those three Coq implementations, we can vaguely check how much additional burden will be added for mechanization when new features are added in programming languages. Table 6.2 shows the vague comparison of Coq implementations for those three languages. As you can see, the size of LJ mechanization is about 9 times larger than that of CFFJ mechanization, and the size of FGJ_Ω_ is about 12 times larger than that of CFFJ.

<table>
<thead>
<tr>
<th>Language</th>
<th>Metatheory</th>
<th>Calculus</th>
<th>Type Safety Proof</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Spec</td>
<td>Proofs</td>
<td>Spec</td>
<td>Proofs</td>
</tr>
<tr>
<td>CFFJ</td>
<td>114</td>
<td>158</td>
<td>164</td>
<td>338</td>
</tr>
<tr>
<td>FBCF</td>
<td>114</td>
<td>158</td>
<td>226</td>
<td>348</td>
</tr>
<tr>
<td>FFMM</td>
<td>136</td>
<td>203</td>
<td>402</td>
<td>1786</td>
</tr>
</tbody>
</table>

Table 6.1: CFFJ, FBCF, and FFMM in Coq

<table>
<thead>
<tr>
<th>Language</th>
<th>Specifications</th>
<th>Proofs</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFFJ</td>
<td>527</td>
<td>496</td>
<td>1023</td>
</tr>
<tr>
<td>LJ</td>
<td>2351</td>
<td>7477</td>
<td>9828</td>
</tr>
<tr>
<td>FGJ_Ω</td>
<td>4904</td>
<td>7380</td>
<td>12284</td>
</tr>
</tbody>
</table>

Table 6.2: CFFJ, LJ, and FGJ_Ω in Coq
Chapter 7. Related Work

Several object oriented languages provide multiple dispatch. Some languages, such as CLOS [6] and Dylan [33], use asymmetric multiple dispatch and asymmetric multiple inheritance to eliminate the ambiguity in method selection at run time.

Some other languages support symmetric multiple dispatch with static rules to check the overloaded method set. Castagna et al. [7] proposed the \( \lambda \& \)-Calculus, an extension of the typed lambda calculus with overloaded functions to present constraints to ensure that there is no ambiguity for each call site. They firstly devised the constraints that can make a hierarchy of method argument types of whole methods in the overloaded method set, which eliminates all possibilities of ambiguous method calls at run time. Similarly, the Fortress team [9] proposed three static rules on the overloaded methods that we mentioned in Section 2.3. Castagna et al. proved the rules guarantee the existence of the most specific method for each method call at run time. However, Castagna et al. did not prove the type safety of any languages with their rules since their purpose was to make language-independent static rules for safe multiple dispatch. FFMM uses three restrictions that are proposed by the Fortress team [9]. Millstein and Chambers designed the Dubious [8], the prototype language that supports modular typing, and symmetric multiple dispatch.

Several researchers have proposed extensions of the Java programming language with symmetric multiple dispatch. Clifton et al. [14] designed MultiJava by adding symmetric multiple dispatch and open classes in Java. Lorenzo et al. [12] proposed Featherweight Java with Multi-methods (FMJ) and proved its type safety. Its overloading rules are similar to our overloading rules. In their next work [13], they proposed a new calculus with multiple dispatch with weaker restrictions that does not have a rule that corresponds to the Meet Rule in Section 2.3. They said that they eliminated the rule to minimize the cost of static type-checking. Lievens and Harrison [16] also proposed a very similar calculus to FMJ, but they included casting expressions that are omitted in FMJ. None of these extensions support multiple inheritance.

The mechanization of type safety proofs for programming languages is not a brand new area any more. Dubois [15] proved the type soundness of ML [22] using CoQ. Fraine et al. [24] proved the type safety of CFFJ, which is similar to FJ, but does not have casting expressions. On top of that, Delaware et al. [21] mechanized the type safety proof of LJ, and Cremet and Altherr mechanized the type safety of FGJ[11, 24], an extension of FGJ with variables representing type constructors. However, none of these mechanizations of Java-like languages provide multiple dispatch or multiple inheritance.
Chapter 8. Conclusion and Future Work

In this paper, we have presented Featherweight Fortress with Multiple Dispatch and Multiple Inheritance, FFMM, a core calculus for the Fortress programming language, which has symmetric multiple dispatch and symmetric multiple inheritance. The calculus formally specifies the static restrictions on valid overloaded method declarations that the Fortress team [9] informally proposed. In FFMM, if the program is well-typed, the type-checker guarantees no undefined method calls and no ambiguous method calls at run time. On top of that, we mechanize the type safety proof of FFMM using a proof assistant tool, COQ. As far as we know, our work is the first mechanized calculus of multiple dispatch in the presence of multiple inheritance. We also strongly believe that our work is adaptable to any Java-like languages with multiple dispatch and multiple inheritance.

We are planning to make a formal calculus by extending FFMM to support other features in Fortress and mechanize its type safety proof. First, we are planning to extend FFMM to permit overloading on generic methods to using the existing idea [23] that the Fortress team proposed recently. Second, to extend the valid overloaded method set, we are planning to add excludes clauses of Fortress to FFMM, which we eliminated in FFMM for simplicity. Finally, we will support the module system that Fortress already has in its implementation, keeping the core expressive power of FFMM.
References


본 논문에서 우리는 Fortress 프로그래밍 언어의 작은 부분을 표현해준 언어임과 동시에 여러 함수 인자의 동적 정보를 참고한 실행 함수 합당 방식과 (multiple dispatch) 다중 상속을 (multiple inheritance) 포함하고 있는 언어인 Featherweight Fortress with Multiple Dispatch and Multiple Inheritance, 즉 FFMM를 설계하였고, 또한 이 언어의 타입 안전성 증명을 CoQ을 이용하여 구현하였다. 여러 함수 인자의 동적 정보를 참고한 실행 함수 합당 방식은 프로그램이 실행되는 시기에 여러 개의 동일한 이름을 가진 함수들 중 실제로 실행할 함수를 선택하는 데 있어서 여러 함수 인자의 실행 시간 타입 정보를 이용하는 방식을 의미하며, 다중 상속은 하나의 타입이 여러 개의 부모 타입을 가질 수 있다는 것을 의미한다. 이 두 가지 개념은 프로그래밍 언어가 높은 표현력을 가질 수 있도록 도와준다. 하지만, 여러 함수 인자를 참고한 실행 함수 합당 방식과 다중 상속을 지원하는 언어들은 실행 시간에 수행되는 함수 호출에 있어서 호출 할 함수가 없는 경우나 (undefined method calls) 여러 개의 함수들 중 어떠한 함수를 호출해야 할지 고르기 애매한 경우를 (ambiguous method calls) 막기 위하여 정적인 제약 사항들을 필요로 한다. 기존에 실행 시간에 호출할 함수가 없는 경우와 어떠한 함수를 호출해야 할지 고르기 애매한 경우가 있거나 아니면하였다는 것을 보장하는 정적인 제약 사항들을 제한한 연구가 있었지만, 그 제약 사항들은 특정한 언어와 밀접하게 연관이 있지는 않았다. 따라서, 우리는 여러 함수 인자의 동적 정보를 참고한 실행 함수 합당 방식과 다중 상속을 안전하게 언어에 포함시키기 위하여 필요한 정적인 제약 사항들을 반영하고 있는 언어를 고안하였으며, 또한 이 언어가 타입 안전성을 만족한다는 것을 엄밀히 보여주기 위하여 우리는 이 언어와 언어의 타입 안전성 증명을 CoQ을 사용하여 구현하였다.
감사의 글

먼저 부족한 저를 항상 이끌어 주시고 저의 모범이 되어 주신 최광무 교수님과 한태숙 교수님께 감사드립니다. 또한, 바쁘신 와중에도 시간을 촉해 논문 심사를 해 주신 최장우 교수님과 한태숙 교수님께도 깊은 감사를 드립니다. 그리고 연구를 진행하는 데 있어서 많은 도움을 주셨던 이계식 교수님, 박성우 교수님, 그리고 종현이에게도 감사드립니다. CoQ와 타입 이론을 처음 접하고 어렵다가는 데 있어서 항상 도움을 주었던 현익이형, 준형이, 그리고 성경이에게도 감사를 드리며, 이 외에도 타입 이론을 함께 공부하고 서로를 겪려해 준 세원이형, 장희, 성훈이, 준수, 재준이에게도 깊은 감사를 드립니다. 많은 분들의 배려와 도움으로 석사 시절동안 많은 것들을 배우고 즐겁게 연구를 할 수 있었습니다. 그리고, 학부 시절 제가 대학원 진학을 하는데 있어 많은 조언을 해 주신 엽영익 교수님과 최형기 교수님께도 깊은 감사를 드립니다. 끝으로, 항상 저를 믿어 주시고 사랑해 주신 부모님과 친혈에게 고맙다는 말씀을 하고 살습니다.
이력서

이름 : 김지웅
E-mail 주소 : gbali@kaist.ac.kr

학력

2002. 3. – 2009. 8. 성균관대학교 정보통신학부 컴퓨터공학과 (B.S.)
2009. 9. – 2011. 8. 한국과학기술원 전산학과 (M.S.)

연구 업적


2. Jieung Kim and Sukyoung Ryu, COQ Mechanization of Featherweight Fortress with Multiple Dispatch and Multiple Inheritance, in preparation.