

Modular Multiple Dispatch with Multiple Inheritance

Eric Allen J.J. Hallett Victor Luchangco Sukyoung Ryu Guy L. Steele Jr.

Sun Microsystems

{eric.allen,joseph.hallett,victor.luchangco,sukyoung.ryu,guy.steele}@sun.com

Abstract

Overloaded functions and methods with multiple dispatch are useful for extending the functionality of existing classes in an object-oriented language. However, such functions introduce the possibility of ambiguous calls that cannot be resolved at run time, and modular static checking that such ambiguity does not exist has proved elusive in the presence of multiple implementation inheritance. We present a core language for defining overloaded functions and methods that supports multiple dispatch and multiple inheritance, together with a set of restrictions on these definitions that can be statically and modularly checked. We have proved that these restrictions guarantee that no undefined nor ambiguous calls occur at run time, while still permitting various kinds of overloading.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features

General Terms Languages

Keywords overloading, multiple dispatch, multiple inheritance, modularity

1. Introduction

Modularity and extensibility are important principles in the design of programs. A modular program is divided into software components, which interoperate through explicit interfaces. Clients of a component depend only on its interface, and thus, the static correctness of a program can be established modularly, by checking that each component correctly implements its interface. An extensible program allows a programmer to augment an existing component without modifying it.

Most object-oriented languages support modular and extensible programming through *classes*. A class defines an object type, which specifies the methods of the object. Classes can be extended to define subclasses of the extended class. Class extension implies both subtyping and inheritance (i.e., the implementation of a method in a superclass is available to the subclass without duplicating the code). In a language providing *multiple inheritance*, a class may reuse code in multiple classes, and an object of that class may be used where an object of any of its superclasses is expected.

One weakness of many object-oriented languages is that all the methods associated with a particular class must be specified with the definition of that class. In such languages, it is not possible

to extend an existing class with entirely new functionality. We can address this weakness by allowing the programmer to define *functions*, independent of any class.

To exploit the type hierarchy of object-oriented languages, it is convenient to allow functions and methods to be *overloaded*: a function or method may have multiple definitions with different parameter types; when an overloaded function or method is invoked, the most specific applicable definition is executed. For example, overriding inherited methods is a limited form of overloading provided by many object-oriented languages. Determining the applicable definitions and which is most specific among them is the responsibility of the *dispatch mechanism*.

There are several choices of dispatch mechanisms: For example, the JavaTM programming language [3] employs a single-dispatch mechanism: a distinguished argument, called the *receiver*, is used for dispatch. (The Java programming language allows overloaded definitions with the same receiver type, but it resolves overloading for other parameters statically.) Other languages such as CLOS [2] use *asymmetric multiple dispatch*, in which the most specific definition is chosen by testing each parameter type for specificity in a specified parameter order (often left to right). However, resolving overloading based on either the static types of the arguments or a particular order of function parameters may be confusing, and can force programmers to reorder a function's parameter list so that the appropriate dispatch semantics is used.

An alternative approach is to use *symmetric multiple dispatch*, in which the run-time types of all the arguments are used to choose the most specific applicable definition; that is, the definition whose parameter types are subtypes of the corresponding parameter types of any other applicable function definition. However, symmetric multiple dispatch introduces the possibility of ambiguity: there may be no definition that is more specific than all other applicable definitions for the given arguments. For example, suppose there are two types A and B, and B is a subtype of A, and there are the following two definitions for a function *f*:

$$\begin{aligned} f(x:A, y:B) &= 1 \\ f(x:B, y:A) &= 2 \end{aligned}$$

Then the call *f(b,b)*, where *b* has type B, is ambiguous: both definitions are applicable, but neither is more specific than the other.

Ambiguity also arises from multiple inheritance. For example, suppose there are additionally types C and D such that C is a subtype of A but not B, and D is a subtype of both B and C. If *g* has definitions:

$$\begin{aligned} g(z:B) &= 3 \\ g(z:C) &= 4 \end{aligned}$$

then the call *g(d)*, where *d* has type D, is ambiguous.

Because ambiguous calls cannot be resolved at run time, restrictions must be imposed at compile time. Instead of rejecting ambiguous calls at call sites, we put restrictions on the set of overloaded definitions. Unfortunately, in object-oriented languages that sup-

```

cd ::= component C
      import * from  $\vec{C}$ 
      end
d ::= td
   | od
   | fd
td ::= trait T extends { $\vec{T}$ } excludes { $\vec{T}$ }  $\vec{md}$  end
od ::= object O( $\vec{x}:\vec{\tau}$ ) extends { $\vec{T}$ }  $\vec{cmd}$  end
fd ::= f( $\vec{x}:\vec{\tau}$ ): $\tau = e$ 
md ::= f( $\vec{mx}$ ): $\tau$ 
      |  $\vec{cmd}$ 
cmd ::= f( $\vec{mx}$ ): $\tau = e$ 
mx ::= x: $\tau$ 
     | self
e ::= x
   | self
   | O( $\vec{e}$ )
   | e.x
   | f( $\vec{e}$ )
 $\tau$  ::= T
      | O

```

Figure 1. Syntax of CF

port the definition of top-level functions (i.e., functions not defined within any class), most such restrictions require whole-program analysis; they cannot be checked modularly. Millstein and Chambers studied this problem and developed *System M*, a set of such restrictions that can be checked modularly. But System M prevents multiple implementation inheritance across module boundaries [9].

In this paper, we describe a new set of overloading restrictions that can be statically and modularly checked without forbidding multiple inheritance. We present these restrictions in the context of *CF*, a distilled version of the Fortress programming language [1], an object-oriented language with support for a generalized form of methods defined within traits [11], and functions defined outside of any trait. Fortress uses symmetric multiple dispatch for both functions and methods, and supports multiple inheritance. Fortress also provides components, which contain declarations of traits, objects and functions. The restrictions presented ensure that no calls will be undefined nor ambiguous at run time.

2. Language

In this section, we describe CF, and establish notation and terminology that we use to describe the restrictions, which are given in the next section. The syntax of CF is provided in Figure 1. The metavariables C ranges over component names; T ranges over trait names; O ranges over object names; f ranges over function and method names; and x ranges over variable and field names. For the sake of brevity, we informally describe the semantics of CF and refer interested readers to the Fortress language specification [1].

Components in CF are the units of compilation, and thus of typechecking. They import signatures of declarations from other components, which serve as “interfaces” of the components. Components declare traits, objects, and functions. Any imported declaration is referred to with its unqualified name.

A trait declaration consists of a header and a set of method declarations. The header specifies the name of the trait, a set of traits to extend, and a set of traits to exclude. A trait *provides* the method declarations that its declaration contains or that are provided by the traits it extends; we say the latter are *inherited* from

the extended traits. A method declaration in a trait may be *abstract* (without a body) or *concrete* (with a body). A trait may provide multiple declarations of a single method; such declarations are *overloaded*. Calls to overloaded method declarations are resolved via symmetric multiple dispatch. A trait is disjoint from any trait it excludes: no trait may extend (directly or indirectly) two traits if one excludes the other.

An object declaration consists of a header and a set of concrete method declarations; it declares a type and a constructor. In addition to the name of the object type, and the extends and excludes clauses, the header specifies a list of fields, which are passed in as parameters to the constructor. An object type cannot be extended, and thus, implicitly excludes any trait that it does not extend (directly or indirectly). An object must provide a concrete declaration for every abstract method declaration it inherits.

For a trait or object types T and U , we say that U is a subtype of T , and write $U \preceq T$, if U extends T or $U = T$. We write $U \prec T$ when $U \preceq T$ and $U \neq T$. We use $U \cap T$ to denote the intersection type of U and T (i.e., the type consisting of all objects that extend both T and U).

As in other object-oriented languages, abstract methods allow the designer of a trait to place constraints on all objects extending the trait: an object extending the trait must provide a concrete implementation of the abstract method. For example, the abstract method *append* declared in a trait *List*:

```

trait List
  append(self, ys : List) : List
  ...
end

```

requires any object extending *List* to provide a concrete method *append*:

```

object Cons(x : Z, xs : List) extends List
  append(self, ys : List) : List = Cons(x, xs.append(ys))
  ...
end

object Empty extends List
  append(self, ys : List) : List = ys
  ...
end

```

Unlike the Java programming language, in which a method invocation is preceded by a receiver expression whose value is bound to an implicit *this* parameter, method invocation in CF uses function call syntax¹ and a method declaration has a unique parameter named *self* at an arbitrary position in its parameter list. When a method is called, the *self* parameter is bound to the argument in its corresponding position. For example, the following are a legal method declaration and a legal method invocation in CF:

```

object Cons(x : Z, xs : List) extends List
  cons(y : Z, self) : List = Cons(y, self)
  ...
end

cons(5, Cons(3, Empty))

```

The static type of the *self* parameter is the type whose declaration contains the method declaration (its *owner*). By allowing the position of the *self* parameter to vary, CF supports better encapsulation of methods. For example, the following overloaded matrix-vector multiply methods

```

trait Vector end

```

¹Fortress also permits method declarations with the conventional “dotted” notation. However, we elide this feature in CF for the sake of brevity.

```

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

```

can be declared within the Matrix trait instead of being split between the Matrix and Vector traits as follows:

```

trait Matrix
  multiply(v : Vector) : Matrix
end

trait Vector
  multiply(m : Matrix) : Matrix
end

```

In addition to methods, CF provides *functions*; whereas a method is declared within a trait or object declaration, a function is declared at the top level of a component. Functions may be overloaded as well. Functions declared in a component may be overloaded with imported functions from another component. Such an overloading does not change the function in the other component; the importing component declares a new set of overloaded functions that has all the imported functions and those added by the component.

3. Overloading Requirements

In this section, we describe how to determine whether a set of overloaded declarations for a function or method is valid (i.e., any call to the function or method can be resolved unambiguously at run time). A component C satisfies the overloading requirements if for every function or method declared in C , the set of declarations for that function or method in C or provided by a component imported by C is valid. The validity of a set of declarations depends only on the signatures of the declarations (i.e. the types of the parameters, where the owner of the declaration is provided as the type of a `self` parameter, and the return type). Therefore, whether a component satisfies the overloading requirements can be checked modularly by looking only at the signatures of declarations in the component and in components it imports.

We determine whether a set of overloaded declarations is valid by independently considering every pair of declarations in that set. In particular, we say that a pair of declarations is a valid overloading if it satisfies one of the rules given below. Informally, a pair of declarations is a valid overloading if they do not have the same signature and for any call to which both declarations are applicable, there is some applicable declaration (possibly one of the pair) that is at least as specific as both declarations.

To state the rules for valid overloading, we use $f(P) : U$ to refer to a declaration of f whose parameter type is P , which may be a tuple type (i.e., there may be more than one parameter), and return type is U . We elide the return type when it is not relevant to the discussion. The subtype relation and the intersection operator on tuple types are defined elementwise. Two tuple types are disjoint (i.e., they exclude each other) if they have different numbers of elements or any pair of corresponding element types are disjoint. For the rest of this section, let $f(P) : U$ and $f(Q) : V$ be two distinct declarations in the same component.

If P and Q are disjoint then $f(P)$ and $f(Q)$ do not introduce ambiguity: they are never applicable to the same call. For example, the following is a valid overloading because `Empty` and `Cons` are disjoint:

```

print(x : Empty) = ...
print(x : Cons) = ...

```

Similarly, the declarations of `multiply` in trait `Matrix` are a valid overloading, because `Matrix` excludes `Vector`. We capture this

with the following rule:

The Exclusion Rule: If P and Q are disjoint types then $f(P)$ and $f(Q)$ are a valid overloading.

Note that declarations with different numbers of parameters are also allowed by the Exclusion Rule because tuple types with different numbers of parameters are disjoint.

If each parameter type of one declaration is a subtype of the corresponding parameter type of the other (and at least one is different), then there is no ambiguity between these two declarations: for any call to which both are applicable, the first is more specific. As in other object-oriented languages, to ensure type safety in the face of dynamic dispatch, we require that the return type of the second declaration is a subtype of the return type of the first.

The Subtype Rule for Functions: If $P \prec Q$ and $U \preceq V$ then $f(P) : U$ and $f(Q) : V$ are a valid overloading.

The same reasoning holds for methods, except the position of the `self` parameter is taken into account.

The Subtype Rule for Methods: If $f(P) : U$ and $f(Q) : V$ have `self` parameters in the same position and $P \prec Q$ and $U \preceq V$ then the declarations are a valid overloading.

If neither the Exclusion Rule nor the Subtype Rule apply, then the declarations introduce the possibility of ambiguity. To avoid this ambiguity, we require a *disambiguating declaration*; that is, for any call to which both declarations are applicable, there must be a third, more specific, declaration that is also applicable. Thus, at run time, neither of the pair of declarations is executed for such a call because the disambiguating declaration is also applicable, and it is more specific than both. For function declarations, we require that there is a single declaration that is applicable whenever both declarations are applicable:

The Meet Rule for Functions: Suppose that neither P nor Q is a subtype of the other. Then $f(P)$ and $f(Q)$ are a valid overloading if there is a declaration $f(P \cap Q)$ declared or imported by that component.

The Meet Rule for Functions requires that the parameter type of the disambiguating declaration to be the intersection of the declarations it disambiguates. For methods, we need a slightly different rule to take into account the `self` parameter. Consider, for example, the following code:

```

trait ℤ
  negative(self): ℤ
end

trait ℝ
  negative(self): ℝ
end

```

In this case, \mathbb{Z} and \mathbb{R} are not disjoint, nor is one the subtype of the other, and there is no disambiguating declaration, and indeed, we cannot define a declaration on the intersection of \mathbb{Z} and \mathbb{R} , because intersection types are not first-class in CF. However, we want to allow this overloading because these declarations are ambiguous only for calls on arguments that extend both \mathbb{Z} and \mathbb{R} , and any type that extends them both can include a new declaration that disambiguates them. To allow such overloading, we define the following Meet Rule for Methods:

The Meet Rule for Methods: Suppose that neither P nor Q is

a subtype of the other. Then $f(P)$ and $f(Q)$ are a valid overloading if

- they have `self` parameters in the same position, and
- any trait or object declaration that provides both $f(P)$ and $f(Q)$ also provides a declaration $f(R)$ with
 - the `self` parameter in the same position, and
 - $R_i = P_i \cap Q_i$ for all positions i other than the position of the `self` parameter, and
 - the owner of $f(R)$ is a subtype of both the owner of $f(P)$ and the owner of $f(Q)$.

The Meet Rule should not be difficult to obey, especially because the compiler can give useful feedback. For example:

```
foo(x : Number, y : Z64) = ...
foo(x : Z64, y : Number) = ...
```

Assuming that $\mathbb{Z}64 \prec \text{Number}$, the compiler reports that these two declarations are a problem because of ambiguity and suggests that a new declaration for $foo(\mathbb{Z}64, \mathbb{Z}64)$ would resolve the ambiguity. As another example, consider:

```
bar(x : Printable) = ...
bar(x : Throwable) = ...
```

Assuming that `Printable` and `Throwable` are not comparable by the subtyping relation, the compiler reports that these two declarations are a problem because `Printable` and `Throwable` are incomparable but possibly overlapping types (i.e., there may be a third type that is a subtype of them both).

The restrictions on overloaded declarations prevent ambiguity from both multiple dispatch and multiple inheritance. By checking declarations for potential ambiguity rather than calls for actual ambiguity, the restrictions account for multiple dispatch. In essence, the restrictions associate the parameter types of a declaration with the dynamic argument types of a call. The restrictions prevent ambiguous calls in the presence of multiple inheritance by requiring the existence of a declaration that is more specific than any two inherited declarations. Moreover, this third declaration must be applicable to any call to which both of the inherited declarations are applicable. We prove that these restrictions are sufficient to guarantee no undefined or ambiguous function calls at run time in Appendix A. The case for methods is analogous.

4. Related Work

Our system builds on the work of Millstein and Chambers [9, 7], who introduce the *Dubious* language to illustrate and explore the challenges of modularly ensuring that no undefined nor ambiguous calls occur at run time, resulting in the definition of *System M*, a statically checkable set of restrictions on modules that guarantees this property.² *Dubious* supports overloaded functions with symmetric multiple dispatch (which they call *multimethods*) and multiple inheritance. Multimethod and type declarations are grouped into modules, and *Dubious* requires every multimethod to have a principal type: the parameter and return types of any definition for a multimethod must be subtypes of their corresponding types in the principal type. System M introduces the notion of an *owner argument position*, and requires that any definition of a multimethod appears either in the module that declares the principal type of the multimethod or in the module that declares the type at the definition’s owner argument position. These cases correspond respectively in CF to functions and to methods for which the `self` pa-

²They also define other sets of restrictions, but these require a link-time check, and so are not truly modular.

parameter is in the same position in all declarations. However, System M disallows multiple implementation inheritance across module boundaries. Also, *Dubious* does not provide exclusion relations between types, and thus cannot support multimethods that take different numbers of arguments or otherwise don’t have a principal type, nor can it allow the position of the owner to vary. In addition, the semantics of overloading across module boundaries in CF and *Dubious* differ slightly: In CF, an overloaded definition for a function imported from another component does not change the function in the imported component; it defines a new function that has all the definitions on the imported function and those added by the importing component. In *Dubious*, there is only one multimethod, which is overloaded with the definitions in all the modules. Thus, calls in one module may use definitions in other modules not imported by that module. In our system, this is possible only for methods, not for functions.

The semantics of *Dubious* has been incorporated into MultiJava [5], an extension of the Java programming language. Relaxed MultiJava[10] is an extension of MultiJava that allows for more overloading than its predecessors at the cost of static checking: it is possible for a function call to be ambiguous; such ambiguities are not caught until class load time.

Lee and Chambers developed F(EML) [6], which improves the module system of EML [8], an ML-like language that incorporates aspects of *Dubious* and System M. CF and F(EML) differ significantly in several ways: First, CF does not support parameterized modules. Second, although F(EML) allows exclusion relations between types, it treats inequality explicitly, tracking the *freshness* of classes and other declarations, which complicates its type system. Third, F(EML) does not provide multiple inheritance.

Castagna [4] has argued that covariant and contravariant type relationships characterize two distinct mechanisms in object-oriented languages, and therefore can be safely integrated together. In our work, the so-called covariance rule manifests itself as the Subtype Rule. And the contravariance rule is the definition of the subtyping relation for function types, which we do not show here. Castagna gives a restriction on overloadings that is very similar to our Meet Rule, however, there is no discussion of modularity, which is one of the main goals of this work.

5. Conclusion and Future Work

We have described an object-oriented language with multiple inheritance and a generalized form of methods to allow greater encapsulation of functionality. The language allows functionality to be added to existing traits (or equivalently, classes) by top-level function definitions. We allow both methods and functions to overload and define a symmetric multiple dispatch semantics. Previous work sacrificed multiple implementation inheritance across module boundaries to achieve modular type checking. We give modular overloading restrictions with multiple inheritance and prove that these restrictions prevent ambiguous or undefined calls at run time.

In the future we plan to integrate user-defined implicit coercion into our language. Coercion may enlarge the set of applicable declarations to a given call. As a result, the restrictions on overloaded declarations will need to take coercion into account. Coercion may also be taken into account when determining more specific relationships between declarations.

Methods and functions with generic type parameters also complicate the overloading restrictions described in this paper. A first approach is to require all overloadings to contain type parameters that are equivalent up to α -renaming. A more general approach may be to allow a subtype relationship between type parameter bounds.

References

- [1] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0 α , Sept. 2006.
- [2] D. G. Bobrow, L. G. DiMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23, Sept. 1988.
- [3] G. Bracha, G. Steele, B. Joy, and J. Gosling. *JavaTM Language Specification, The 3rd Edition (Java Series)*. Addison-Wesley Professional, July 2005.
- [4] G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.
- [5] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3), May 2006.
- [6] K. Lee and C. Chambers. Parameterized modules for classes and extensible functions. In *Proceedings of the 20th European Conference on Object-Oriented Programming*. Springer-Verlag, 2006.
- [7] T. Millstein. *Reconciling Software Extensibility with Modular Program Reasoning*. PhD thesis, University of Washington, 2003.
- [8] T. Millstein, C. Bleckner, and C. Chambers. Modular typechecking for hierarchically extensible datatypes and functions. *ACM Transactions on Programming Languages and Systems*, 26(5):836–889, Sept. 2004.
- [9] T. Millstein and C. Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, May 2002.
- [10] T. D. Millstein, M. Reay, and C. Chambers. Relaxed multijava: balancing extensibility and modular typechecking. In R. Crocker and G. L. S. Jr., editors, *OOPSLA*, pages 224–240. ACM, 2003.
- [11] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings of the 17th European Conference on Object-Oriented Programming*. Springer, July 2003.

A. Proof of Overloading Resolution for Functions

We prove that the restrictions placed on overloaded declarations are sufficient to guarantee no undefined nor ambiguous calls at run time (the case for methods is analogous). We use *static call* $f(A)$ to refer to a call with name f and whose argument has static type A , and *dynamic call* $f(X)$ to refer to a call with name f and whose argument, when evaluated, has dynamic type X . We write $U \diamond T$ when U excludes T .

Consider a static call $f(A)$ and its corresponding dynamic call $f(X)$. Let Δ be the set of parameter types of declarations of f that are applicable to the dynamic call $f(X)$. Let Σ be the set of parameter types of declarations of f that are applicable to the static call $f(A)$. Moreover, let σ and δ be the subset of Σ and Δ for which no type in Σ and Δ is more specific, respectively:

$$\begin{aligned} \sigma &= \{S \in \Sigma \mid \neg \exists S' \in \Sigma : S' \prec S\} \\ \delta &= \{D \in \Delta \mid \neg \exists D' \in \Delta : D' \prec D\}. \end{aligned}$$

Below we prove:

$$|\Sigma| \neq 0 \text{ implies } |\sigma| = 1, \quad \text{and} \quad |\Sigma| \neq 0 \text{ implies } |\delta| = 1.$$

Informally, if any declaration is applicable to a static call then there exists a single most specific declaration that is applicable to the static call and a single most specific declaration that is applicable to the corresponding dynamic call.

LEMMA 1. *Given an acyclic, irreflexive binary relation R on a set S , and a finite nonempty subset A of S , the set $\{a \in A \mid \neg \exists a' \in A : (a', a) \in R\}$ is nonempty.*

PROOF: Consider the relation R on S as a directed acyclic graph. Let A represent a subgraph. Then the Lemma amounts to proving that there exists a node in the graph represented by A with no edges pointing to it. This follows from the fact that A is finite and the graph is acyclic.

LEMMA 2. $\Sigma \subseteq \Delta$.

PROOF: Notice that $X \preceq A$ by type soundness. If $f(P)$ is applicable to the call $f(A)$ then $A \preceq P$. Notice that $X \preceq A$ implies $X \preceq P$. Therefore $f(P)$ is applicable to the call $f(X)$.

LEMMA 3. *If $|\Delta| \geq 1$ then $|\delta| \geq 1$. Also, if $|\Sigma| \geq 1$ then $|\sigma| \geq 1$.*

PROOF: Follows from Lemma 1 where S is the set of all types, A is Δ and Σ respectively, and the relation \prec is acyclic and irreflexive.

LEMMA 4. *If $|\Sigma| \geq 1$ then $|\delta| \geq 1$.*

PROOF: Follows from Lemmas 2 and 3.

LEMMA 5. $|\sigma| \leq 1$. Also, $|\delta| \leq 1$.

PROOF: We prove this for δ , but the case for σ is identical. For the purpose of contradiction suppose there are two declarations $f(P)$ and $f(Q)$ in δ . Since both $f(P)$ and $f(Q)$ are applicable to the call $f(X)$ we have $X \preceq P \cap Q$. By the overloading restrictions, $P \neq Q$ and either $P \diamond Q$ or there is a declaration $f(P \cap Q)$ in the same component. Since it cannot be the case that $P \diamond Q$ there must exist a declaration $f(P \cap Q)$ in the same component. Since $P \cap Q \preceq P$ and $P \cap Q \preceq Q$ we know $f(P \cap Q)$ is applicable to the call $f(X)$. Since $P \neq Q$ either $P \cap Q \prec P$ or $P \cap Q \prec Q$. Either case contradicts our assumption.

THEOREM 1. *If $|\Sigma| \neq 0$ then $|\sigma| = 1$. If $|\Sigma| \neq 0$ then $|\delta| = 1$.*

PROOF: Follows from Lemmas 3, 4 and 5.

THEOREM 2. *If $\sigma = \{S\}$ and $\delta = \{D\}$ then $D \preceq S$.*

PROOF: If the declaration with parameter type S and the declaration with parameter type D satisfy the Subtype Rule then the theorem is proved. Otherwise, by the definition of σ we have $\sigma \subseteq \Sigma$. Therefore $S \in \Sigma$. By Lemma 2, $S \in \Delta$. Notice that $S, D \in \Delta$ implies $X \preceq S$ and $X \preceq D$. Therefore, $S \not\prec D$. By the More Specific Rule for Functions, there must exist a declaration with parameter type $S \cap D$. Because $X \preceq (S \cap D)$, $(S \cap D) \in \Delta$. Notice $(S \cap D) \preceq S$ and $(S \cap D) \preceq D$. By the definition of δ , we have $\neg \exists D' \in \Delta : D' \prec D$. In particular, $(S \cap D) \not\prec D$. Therefore $(S \cap D) = D$.