Polymorphic Symmetric Multiple Dispatch with Variance

GYUNGHEE PARK, KAIST, South Korea and Oracle Labs, USA

JAEMIN HONG, KAIST, South Korea

GUY L. STEELE JR., Oracle Labs, USA

SUKYOUNG RYU, KAIST, South Korea

Many object-oriented languages provide *method overloading*, which allows multiple method declarations with the same name. For a given method invocation, in order to choose what method declaration to invoke, *multiple dispatch* considers the run-time types of the arguments. While multiple dispatch can support binary methods (such as mathematical operators) intuitively and consistently, it is difficult to guarantee that calls will be neither ambiguous nor undefined at run time, especially in the presence of expressive language features such as *multiple inheritance* and *parametric polymorphism*. Previous efforts have formalized languages that include such features by using *overloading rules* that guarantee a unique and type-sound resolution of each overloaded method call; in many cases, such rules resolve ambiguity by treating the arguments asymmetrically. Here we present the first formal specification of a strongly typed object-oriented language with *symmetric* multiple dispatch, multiple inheritance, and parametric polymorphism with *variance*. We define both a static (type-checking) semantics and a dynamic (dispatching) semantics and prove the type soundness of the language, thus demonstrating that our novel dynamic dispatch algorithm is consistent with the static semantics. Details of our dynamic dispatch algorithm address certain technical challenges that arise from structural asymmetries inherent in object-oriented languages (e.g., classes typically declare ancestors explicitly but not descendants).

CCS Concepts: • **Software and its engineering** → **General programming languages**; • **Social and professional topics** → *History of programming languages*;

Additional Key Words and Phrases: Method Overloading, Symmetric Multiple Dispatch, Parametric Polymorphism, Variance

ACM Reference Format:

Gyunghee Park, Jaemin Hong, Guy L. Steele Jr., and Sukyoung Ryu. 2019. Polymorphic Symmetric Multiple Dispatch with Variance. *Proc. ACM Program. Lang.* 3, POPL, Article 11 (January 2019), 28 pages. <https://doi.org/10.1145/3290324>

1 INTRODUCTION

In object-oriented languages, *method overloading* and its unique resolution are important features. Method overloading allows multiple method declarations of the same name but different parameter types. Thus, it enables to define different behaviors of `+` or `=` for different parameter types, for example. For a given overloaded method invocation of `equal`, a resolution mechanism should select a single declaration to call among the overloaded declarations of `equal` using the argument types of the method invocation.

When resolving an overloaded method invocation, *dynamic dispatch* considers the run-time types of the arguments of the method invocation. When a dispatch mechanism considers the run-time

Authors' addresses: Gyunghee Park, KAIST, South Korea, Oracle Labs, USA; Jaemin Hong, KAIST, South Korea; Guy L. Steele Jr. Oracle Labs, USA; Sukyoung Ryu, KAIST, South Korea.

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

© 2019 Copyright held by the owner/author(s).

2475-1421/2019/1-ART11

<https://doi.org/10.1145/3290324>

type of only a single argument, usually the receiver of a method call, as in languages like C++ and Java, it is *single dispatch*. By contrast, *multiple dispatch* considers the run-time types of more than one argument as in languages like Common Lisp, Fortress, and Julia. Especially, *symmetric multiple dispatch* considers multiple arguments equally, which supports mathematical operators like binary methods intuitively and uniformly [Clifton et al. 2000].

While method overloading and its resolution by dynamic dispatch give a consistency in naming methods that logically perform very similar tasks, they impose two restrictions to guarantee a unique and type-sound resolution of each overloaded method invocation [Castagna et al. 1992, 1995]. First, the dispatch mechanism should be able to find the *most specific applicable* method declaration for any overloaded method invocation. A method declaration is applicable to given arguments if its parameter types are supertypes of the types of the arguments. One method declaration is more specific than another if the *domain type* of the former is a subtype of that of the latter. Hereafter, we consider that every method declaration has a single parameter possibly a tuple of two or more components, and call the parameter type of a method declaration its domain type. Second, for a given overloaded method invocation, its static type should be a supertype of the return type of a dynamically chosen method declaration by the dispatch mechanism. Otherwise, a method invocation result may be used in a type-unsound manner.

Moreover, expressive language features often add various possibilities to make overloaded method invocations ambiguous. As we discuss below, such features make it difficult to design a language that guarantees no ambiguous nor undefined calls at run time.

One such a feature is *multiple inheritance*, which allows one class to extend multiple classes. While multiple inheritance supports modular and reusable code, it easily introduces ambiguous method calls. Because one parent class may not be more specific than another parent class, dynamically selecting the most specific applicable method declaration may be unclear. For example, consider that each of two classes A and B has a method declaration m with the same parameter type `Number`, class C extends both A and B , and c is an instance of C . Then, evaluation of $c.m(0)$ results in an ambiguous call because neither A nor B is more specific than each other.

Another feature is *parametric polymorphism*, or *generics*. By introducing type parameters in class and method declarations, parametric polymorphism enables programmers to manage types in a fine-grained way. For example, the `List` class may be instantiated as either `List[String]` or `List[Integer]`. However, selecting the most specific applicable declaration among polymorphic method declarations at run time is not trivial. Moreover, many languages with parametric polymorphism support automatic inference for the type arguments of polymorphic method invocations; at run time, dynamic dispatch must not only find the most specific applicable declaration, but also infer appropriate type arguments for that declaration.

Finally, *variance* defines additional subtype relations for polymorphic types [Igarashi and Viroli 2006]. Without variance, instances of the same polymorphic type are in a subtype relation only when their type arguments are the same type. Variance supports subtype relationships between different instances of the same polymorphic type by allowing the programmer to label each type parameter of a class *covariant*, *contravariant*, or *invariant*. If a parametric type T has one covariant type parameter, then it preserves the subtype relation of its type argument; hence, when a type P is a subtype of another type Q , which we write $P <: Q$, it implies $T[P] <: T[Q]$. If the type parameter of T is contravariant, then T reverses the subtype relationship of its type arguments; hence, $P <: Q$ implies $T[Q] <: T[P]$. If the type parameter of T is invariant, then $T[P] <: T[Q]$ only if $P \equiv Q$, where $P \equiv Q$ denotes that P and Q are equivalent, that is, P is a subtype of Q and vice versa. Variance annotations allow the programmer to define more expressive polymorphic methods. For example, consider the following excerpt from the `List` class source code in Scala [EPFL 2017]:

```
sealed abstract class List[+A] extends Seq[A] {
  ...
  def ::[B >: A](prefix: List[B]): List[B] = ...
  ...
}
```

The `List` class takes a single covariant (denoted by `+`) type parameter `A`, and the `::` method concatenates two lists, its receiver and argument. If `A` were invariant, the element type `B` of the argument list should be a supertype of the element type `A` of the receiver list. However, since `A` is covariant, the method can take a list of any element type `C` that is less than or equal to `B`, and return a list whose element type is `B`, which is a common supertype of the element types of its receiver and argument: $A \leq B$ and $C \leq B$. As one might expect, variance also increases the difficulty of overloading resolution.

Therefore, supporting symmetric multiple dispatch in the presence of various expressive language features has been a long-standing effort. Previous efforts have formalized a variety of core calculi that capture some of the aforementioned features in a piecemeal rather than holistic fashion. While they are widely different in terms of the expressivity, completeness, decidability, and correctness of the formalization of the target languages and their type systems, they all define a set of *overloading rules* that overloaded method declarations should satisfy to make dynamic dispatch select the most specific applicable method declaration with a type-safe return type.

For a simple object-oriented language with multiple inheritance, [Allen et al. \[2008, 2007\]](#) and [Kim and Ryu \[2011\]](#) presented three overloading rules: Exclusion Rule, Subtype Rule, and Meet Rule. Every pair of method declarations with the same name must satisfy *one* of these overloading rules. The authors proved that a set of overloaded method declarations satisfying the overloading rules does not produce any ambiguous method invocation at run time.

To understand the interplay between symmetric multiple dispatch and parametric polymorphism, [Allen et al. \[2011\]](#) focused on a type system without any specific target language and informally presented three other overloading rules: a No Duplicates Rule, a revised Meet Rule, and a Return Type Rule, where every pair of method declarations with the same name must satisfy *all three* overloading rules. The authors showed that interpreting a polymorphic method declaration as a set of infinitely many monomorphic declarations is not appropriate for overloading resolution. Instead, they considered a polymorphic method declaration as a single declaration whose domain type is existentially quantified over its type parameters as proposed by [Bourdoncle and Merz \[1997\]](#).

In this paper, we present a formal core calculus *FGFV* (Featherweight Generic Fortress with Variance) for an object-oriented language with symmetric multiple dispatch, multiple inheritance, and parametric polymorphism with variance. For conciseness, *FGFV* omits functional methods [[Allen et al. 2007](#)] and declarations of closed types [[Allen et al. 2011](#)], but these could be added back to *FGFV* in an obvious way. Unlike the prior work, we formally present overloading rules for *FGFV*, define its dynamic semantics with a novel dynamic dispatch algorithm, and prove type soundness. As we discuss in Section 5.4, defining a provable system for an object-oriented language with asymmetric features requires careful tradeoffs between static expressivity and dynamic efficiency.

In the rest of the paper, we first describe the previous efforts in more detail with concrete code examples (Section 2). After presenting the *FGFV* calculus that represents an object-oriented language with symmetric multiple dispatch, multiple inheritance, parametric polymorphism, and variance (Section 3), we formally define the overloading rules for *FGFV* (Section 4). Then, we define both static and dynamic overloading resolution with a novel dynamic dispatch algorithm, and sketch proofs of the type soundness of *FGFV* (Section 5). Full proofs are available in a companion report [[Park et al. 2018](#)]. Finally, we discuss related work (Section 6) and conclude (Section 7).

2 MOTIVATION

In this section, we first discuss existing research achievements in the pursuit of formally defining object-oriented languages with symmetric multiple dispatch, multiple inheritance, and parametric polymorphism with variance. Then, we present how FGFV completes the long-standing endeavor by defining a core calculus with static overloading rules and a dynamic dispatch algorithm that are proven type-sound.

2.1 Overloading Rules for Multiple Inheritance

As we discussed in Section 1, both symmetric multiple dispatch and multiple inheritance may cause ambiguous method calls that cannot be resolved at run time. Let us assume that class C is the only subtype of both classes A and B , and they are all distinct throughout this section. Consider the following overloaded method declarations of m_1 :

$$m_1(x: B, y: C): B = \dots \qquad m_1(x: C, y: B): B = \dots$$

which have the same return type B but different domain types (B, C) and (C, B) . For a method invocation $m_1(c, c)$ where the run-time type of c is C , it is ambiguous which declaration to invoke due to symmetric multiple dispatch: both method declarations are applicable to (c, c) since both (B, C) and (C, B) are supertypes of (C, C) , and neither declaration is more specific than the other. Also, consider the following overloaded declarations of m_2 :

$$m_2(x: A): \text{Boolean} = \dots \qquad m_2(x: B): \text{Boolean} = \dots$$

which have domain types A and B , respectively. For a method invocation $m_2(c)$, it is ambiguous which declaration to invoke due to multiple inheritance: both declarations are applicable to c and neither declaration is more specific than the other.

In order to guarantee that no such ambiguous method calls are possible at run time, researchers have proposed several calculi with restrictions on overloaded method declarations. CF [Allen et al. 2007], Core Fortress, is a core calculus of the Fortress programming language [Allen et al. 2008], which contains components, top-level functions, functional methods, and declarations of exclusive types in addition to multiple inheritance. However, it does not specify the language semantics and informally provides rules for valid overloaded method declarations. FFMM [Kim and Ryu 2011], Featherweight Fortress with Multiple Dispatch and Multiple Inheritance, simplified CF to focus only on multiple inheritance, and it has a complete formalization of the language semantics including overloading rules and its type soundness proof in Coq. Since the overloading rules of CF and FFMM are the same, we will refer to these two languages collectively as *Featherweight Fortress (FF)*.

The overloading rules of FF require every pair in a set of overloaded method declarations to satisfy *one* of the following rules:

Exclusion Rule: Their domain types are *exclusive* (their intersection is Bottom, where the Bottom type is a subtype of every type and no value belongs to Bottom).

Subtype Rule: The domain type of one declaration is a strict subtype of the domain type of the other, and the return type of the former is a subtype of the return type of the latter.

Meet Rule: The domain types are not exclusive nor in a subtype relation, and the overloaded set contains a third disambiguating method declaration which we call the *meet* of the pair. The meet of two method declarations is more specific than each of the declarations and less specific than any other method declaration that is more specific than both declarations.

Let us revisit the previous examples, with an additional method declaration each for m_1 and m_2 :

$$\begin{array}{lll} d_1^1 : m_1(x: B, y: C): B = \dots & d_1^2 : m_1(x: C, y: B): C = \dots & d_1^3 : m_1(x: C, y: C): C = \dots \\ d_2^1 : m_2(x: A): \text{Boolean} = \dots & d_2^2 : m_2(x: B): \text{Boolean} = \dots & d_2^3 : m_2(x: C): \text{Boolean} = \dots \end{array}$$

Fig. 1. Overloading rules for multiple inheritance and generics where circles denote applicable sets of generic method declarations and pentagons denote instantiated method declarations

Note that, for both m_1 and m_2 , the third method declaration is the meet of the first two declarations. Each set of overloaded method declarations is valid since every pair satisfies one of the overloading rules (for $i = 1$ and 2 , d_i^1 and d_i^2 satisfy the Meet Rule; d_i^1 and d_i^3 satisfy the Return Type Rule, as do d_i^2 and d_i^3).

The overloading rules guarantee two important properties: (1) no ambiguous method calls can occur at run time, and (2) types are preserved even when a method call is dispatched to a different method declaration at run time from a method declaration chosen at compile time. In FF, Exclusion Rule is for the trivial cases where no ambiguous calls can occur, Meet Rule ensures the existence of disambiguating method declarations, and Subtype Rule serves a key role for type preservation.

2.2 Overloading Rules for Multiple Inheritance and Generics

Similarly for multiple inheritance, generics may cause ambiguous method calls that cannot be resolved at run time. Consider the following overloaded declarations of *sort*:

$$\begin{aligned} \text{sort} \llbracket P <: A \rrbracket (x: \text{List}^I \llbracket P \rrbracket): \text{SortedList}^I \llbracket P \rrbracket &= \dots \\ \text{sort} \llbracket P <: B \rrbracket (x: \text{List}^I \llbracket P \rrbracket): \text{SortedList}^I \llbracket P \rrbracket &= \dots \end{aligned}$$

Their domain types are respectively “ $\text{List}^I \llbracket P \rrbracket$ where P is a subtype of A ” and “ $\text{List}^I \llbracket P \rrbracket$ where P is a subtype of B ”; for a method invocation $\text{sort}(l)$ where the run-time type of l is $\text{List}^I \llbracket C \rrbracket$, both declarations are applicable to l and neither declaration is more specific than the other.

In order to support generics as well for symmetric multiple dispatch, [Allen et al. \[2011\]](#) proposed new overloading rules for given generic declarations of types and methods. Even though they did not present a specific language with generics, we refer to the underlying language as *FGF*, Featherweight Generic Fortress. To support generics, the authors re-defined the terms *applicability* and *specificity* for generic method declarations and their overloading rules. A generic method declaration d is applicable to a type α if there exists an instance D of d that is applicable to α , that is, the domain type of D is a supertype of α . For example, consider the following generic method declaration:

$$m \llbracket P <: A \rrbracket (x: P): P = \dots$$

which has one type parameter P with an upper bound A . It is applicable to *Bottom*, C , and A , because they are subtypes of A . In general, a generic method declaration d_1 is more specific than d_2 if the applicable set of d_1 is a subset of that of d_2 , where the *applicable set* of a method declaration is a set of types to which the declaration is applicable. Note that a monomorphic method declaration is a generic method declaration without any type parameter.

The overloading rules of FGF require every pair in a set of overloaded method declarations to satisfy *all* of the following rules as illustrated in Figure 1:

No Duplicates Rule: Distinct method declarations must have different applicable sets (there must be some argument type α for which one method is applicable and the other is not).

Meet Rule: Let S be the intersection of the applicable sets of the two method declarations; if S is not empty, then the overloaded set must contain a method declaration (not necessarily distinct from the two given method declarations) whose applicable set is precisely S .

Return Type Rule: If one method declaration d_1 is more specific than d_2 , for any non-Bottom type α in the applicable set of d_1 , the return type of an instance of d_2 that is applicable to α must be a supertype of the return type of some instance of d_1 that is applicable to α .

Let us revisit the above ambiguous *sort* example with an additional method declaration:

$$\begin{aligned} d_1 : & \quad \text{sort}[P <: A](x: \text{List}^I[P]): \text{SortedList}^I[P] = \dots \\ d_2 : & \quad \text{sort}[P <: B](x: \text{List}^I[P]): \text{SortedList}^I[P] = \dots \\ d_3 : & \quad \text{sort}[P <: C](x: \text{List}^I[P]): \text{SortedList}^I[P] = \dots \end{aligned}$$

This set of overloaded method declarations is valid. For all pairs, the No Duplicates Rule is trivially satisfied. In the case of d_1 and d_2 , the Meet Rule is satisfied (d_3 is their meet), and the Return Type Rule is vacuously satisfied since neither is more specific than the other. For d_1 and d_3 , the Meet Rule is satisfied with d_3 itself as the meet; to see that the Return Type Rule is satisfied, note that List^I is an invariant type constructor since FGF does not support variance. For any $\text{List}^I[X]$ in the applicable set of d_3 , there is exactly one instance of d_1 (with X substituted for P) applicable to it:

$$\text{sort}(x: \text{List}^I[X]): \text{SortedList}^I[X] = \dots$$

Since the return type of this instance equals the return type of the corresponding instance of d_3 :

$$\text{sort}(x: \text{List}^I[X]): \text{SortedList}^I[X] = \dots,$$

d_1 and d_3 satisfy the Return Type Rule. Similarly, d_2 and d_3 also satisfy all of the overloading rules.

Again, the overloading rules ensure that there is no ambiguous method calls at run time, and that types are preserved. In FGF, No Duplicates Rule rules out the trivial ambiguous calls with duplicate declarations. Meet Rule plays an important role for ensuring the existence of a disambiguating method declaration, and Return Type Rule serves a key role for type preservation.

2.3 Overloading Rules for Multiple Inheritance and Generics with Variance

Variance, as well as symmetric multiple dispatch, multiple inheritance, and generics, may cause ambiguous method calls that cannot be resolved at run time. We revisit the ambiguous *sort* example in Section 2.2. This time, we define the same method in a simpler way using covariant definitions of List^C and SortedList^C . Consider the following overloaded method declarations of *sort*:

$$\begin{aligned} d_1 : & \quad \text{sort}(x: \text{List}^C[A]): \text{SortedList}^C[A] = \dots \\ d_2 : & \quad \text{sort}(x: \text{List}^C[B]): \text{SortedList}^C[B] = \dots \end{aligned}$$

Similarly to the example in Section 2.2, for a method invocation $\text{sort}(l)$ where the run-time type of l is $\text{List}^C[C]$, it is ambiguous which declaration to invoke: both d_1 and d_2 are applicable to l and neither is more specific than the other. In order to ensure that there is no ambiguous method calls for *sort* at run time, we need to add a disambiguating method declaration d_3 :

$$d_3 : \quad \text{sort}(x: \text{List}^C[C]): \text{SortedList}^C[C] = \dots$$

with a more specific return type than those of d_1 and d_2 for type preservation.

Variance also adds complications in type-sound dynamic dispatch. Let us now assume that List^C is covariant in its type argument but SortedList^I is invariant. Then, the set of three method declarations above replacing SortedList^C with SortedList^I would be invalid, because the return type $\text{SortedList}^I[A]$ of d_1 would not be a supertype of the return type $\text{SortedList}^I[C]$ of d_3 .

Supporting variance in a type-sound manner requires keeping track of the static types of method invocations at run time. Let us again assume that List^C is covariant but SortedList^I is invariant, and consider the following overloaded set (note that d_3 now has a type parameter P):

$$\begin{aligned} d_1 : & \quad \text{sort}(x: \text{List}^C[A]): \text{SortedList}^I[A] = \dots \\ d_2 : & \quad \text{sort}(x: \text{List}^C[B]): \text{SortedList}^I[B] = \dots \\ d_3 : & \quad \text{sort}[P](x: \text{List}^C[C]): \text{SortedList}^I[P] = \dots \end{aligned}$$

as well as the following generic method declaration of *merge*:

$$\text{merge}[P](x: \text{SortedList}^I[P], y: \text{SortedList}^I[P]): \text{SortedList}^I[P] = \dots$$

For a method invocation $\text{merge}(\text{sort}(l_1), \text{sort}(l_2))$ where the static types of l_1 and l_2 are both $\text{List}^C[A]$ and their run-time types are $\text{List}^C[A]$ and $\text{List}^C[C]$, respectively, at compile time, both method invocations $\text{sort}(l_1)$ and $\text{sort}(l_2)$ choose d_1 and their static types are both $\text{SortedList}^I[A]$. Thus, the above *merge* call statically invokes the following instance of the generic *merge* declaration:

$$\text{merge}(x: \text{SortedList}^I[A], y: \text{SortedList}^I[A]): \text{SortedList}^I[A]$$

that is applicable to $(\text{SortedList}^I[A], \text{SortedList}^I[A])$. However, at run time, while $\text{sort}(l_1)$ chooses d_1 and its return type is $\text{SortedList}^I[A]$, $\text{sort}(l_2)$ chooses d_3 and its return type depends on what actual type is inferred at run time for the type parameter P . If the dynamic type inference algorithm instantiates P as A , $\text{sort}(l_2)$ has type $\text{SortedList}^I[A]$, which is type sound. But it could equally plausibly instantiate P as some other type T , and then no dynamic instance of the generic *merge* declaration would be applicable to $(\text{SortedList}^I[A], \text{SortedList}^I[T])$. Hence, the *merge* call would fail, despite the fact that the program successfully passed static type checking. In order to help run-time type inference algorithms make type-sound choices of type arguments, the language semantics needs to make the static types of method invocations available at run time.

In this paper, we present FGFV, which sheds light on the long-standing endeavor to support strong types, symmetric multiple dispatch, multiple inheritance, and parametric polymorphism with variance. We formalize the static overloading rules for FGFV using existential types and universal types quantified over method type parameters, and design a new dynamic dispatch algorithm that is proven type-sound to guarantee unambiguous method calls and type preservation at run time.

3 LANGUAGE

In this section, we define the abstract syntax of FGFV and its well-formedness requirements.

3.1 Notation

Following, for example, Vytiniotis et al. [2013] and Coughlin and Chang [2014], we use an overline above mathematical material to indicate an appropriately punctuated sequence of zero or more repetitions of the material, with a subscript added to each metavariable in the material. Thus $\{\bar{\alpha}\}$ is equal to $\{\alpha_1, \alpha_2, \dots, \alpha_n\}$. Sometimes we write $\bullet$ as an explicit indication of an empty sequence. Overlines may be nested, in which case they are expanded outermost first, so “ $\{\bar{\{\bar{\xi}\}} <: Q <: \{\bar{\xi}\}\}$ ” expands first to “ $\{\{\bar{\xi}_1\} <: Q_1 <: \{\bar{\xi}_1\}, \{\bar{\xi}_2\} <: Q_2 <: \{\bar{\xi}_2\}, \dots, \{\bar{\xi}_n\} <: Q_n <: \{\bar{\xi}_n\}\}$ ” and then to

$$\begin{aligned} & \{\{\xi_{11}, \xi_{12}, \dots, \xi_{1k_1}\} <: Q_1 <: \{\xi_{11}, \xi_{12}, \dots, \xi_{1m_1}\}, \\ & \{\xi_{21}, \xi_{22}, \dots, \xi_{2k_2}\} <: Q_2 <: \{\xi_{21}, \xi_{22}, \dots, \xi_{2m_2}\}, \\ & \dots, \\ & \{\xi_{n1}, \xi_{n2}, \dots, \xi_{nk_n}\} <: Q_n <: \{\xi_{n1}, \xi_{n2}, \dots, \xi_{nm_n}\} \end{aligned}$$

(conceptually all on one line, broken here for reasons of space). The copies of each inner overline expand independently, so the sequences into which they expand may have different lengths; thus “ $\{\bar{\{\bar{\xi}\}} <: Q <: \{\bar{\xi}\}\}$ ” could expand into, e.g., “ $\{\{\xi_{11}, \xi_{12}\} <: Q_1 <: \{\xi_{11}\}, \{\} <: Q_2 <: \{\xi_{21}, \xi_{22}, \xi_{23}\}\}$ ”.

Program	$\Pi ::= \overline{\psi}, e$
Class declaration	$\psi ::= \text{trait } T[\overline{V} \overline{\beta}] <: \{\overline{t}\} \overline{\mu} \text{ end} \mid \text{object } O[\overline{\beta}](\overline{x}:\overline{\tau}) <: \{\overline{t}\} \overline{\mu} \text{ end}$
Method definition	$\mu ::= m[\overline{\kappa}](\overline{x}:\overline{\tau}): \tau = e$
Class type parameter binding	$\beta ::= P <: \{\overline{\tau}\}$
Method type parameter binding	$\kappa ::= \{\overline{\tau}\} <: P <: \{\overline{\tau}\}$
Variance mark	$V ::= + \mid - \mid =$
Expression	$e ::= z \mid ((\overline{x}:\overline{\tau}): \tau \Rightarrow e) \mid e @ (\overline{e}) \mid O[\overline{\tau}](\overline{e}) \mid e.m(\overline{e})$
Bindable variable	$z ::= x \mid \text{self}$
Type	$\tau ::= P \mid c \mid (\overline{\tau}) \mid (\tau \rightarrow \tau) \mid \text{Any}$
Constructed type	$c ::= t \mid O[\overline{\tau}]$
Trait type	$t ::= T[\overline{\tau}]$

Fig. 2. Abstract syntax of FGFV

Following a suggestion of Steele [2017, 0:59:37], we use an underline below submaterial within overlined material to prevent an overline from attaching subscripts to the metavariables in the submaterial; the effect is to require the underlined material to be the same in each copy. Thus “ $\overline{\Delta \vdash \psi \text{ ok}}$ ” represents the sequence “ $\Delta \vdash \psi_1 \text{ ok} \quad \Delta \vdash \psi_2 \text{ ok} \quad \cdots \quad \Delta \vdash \psi_n \text{ ok}$ ” for any $n \geq 0$. Similarly $\overline{\underline{\Delta} \vdash \xi \text{ ok}}$ expands to “ $\underline{\Delta'} \vdash \xi_{11} \text{ ok} \quad \underline{\Delta'} \vdash \xi_{21} \text{ ok} \quad \cdots \quad \underline{\Delta'} \vdash \xi_{n1} \text{ ok}$ ” and then to

$$\begin{array}{ccccccc}
 \Delta' \vdash \xi_{11} \text{ ok} & \Delta' \vdash \xi_{12} \text{ ok} & \cdots & \Delta' \vdash \xi_{1 m_1} \text{ ok} \\
 \Delta' \vdash \xi_{21} \text{ ok} & \Delta' \vdash \xi_{22} \text{ ok} & \cdots & \Delta' \vdash \xi_{2 m_2} \text{ ok} \\
 \cdots & & & \\
 \Delta' \vdash \xi_{n1} \text{ ok} & \Delta' \vdash \xi_{n2} \text{ ok} & \cdots & \Delta' \vdash \xi_{n m_n} \text{ ok}
 \end{array}$$

The overline notation may explicitly bind index variables by using a range expression. In such cases, the index variables may appear explicitly in the covered material, and subscripts are not automatically attached to metavariables. For example, $(\overline{m(e_i)})^{3 \leq i \leq 6}$ means $(m(e_3), m(e_4), m(e_5), m(e_6))$.

Metavariables are expanded *after* overlines. Multiple occurrences of a metavariable with the same subscript(s) must have the same expansion, but occurrences of a metavariable with different subscript(s) may be expanded independently. Thus “ $\overline{\{\{\overline{\xi}\} <: Q <: \{\overline{\xi}\}\}}$ ” might ultimately expand into, e.g., “ $\{\{A, B\} <: U <: \{\text{List}[\text{Boolean}]\}, \{\} <: V <: \{\text{Any}, C, \text{Number}\}\}$ ”.

We use horizontal lines as overlines and underlines, but also in the usual Gentzen notation for inference rules, with premises above the line and the conclusion below. To help reduce confusion: (1) we use a slightly heavier and wider line for inference rules; (2) the line for inference rules has more space above and below it; (3) we do not use the inference rule notation for axioms (rules that have no premises); and (4) we do not nest the inference rule notation to present derivation trees.

3.2 Syntax

Figure 2 shows the syntax of the FGFV calculus. Metavariables T and U range over trait names; O ranges over object names; x ranges over variable and field names; z is either x or the keyword `self`; m ranges over method names; P, Q, S range over type parameter names; and τ, ω, ζ, ξ range over types.

A program consists of a sequence of class declarations and a top-level expression. A class is either a trait or an object; traits are like Java interfaces that do not have any fields, and objects are like Java final classes that cannot be extended. A trait may have type parameters with variance annotations and upper bounds, extend any number of other traits, and have method definitions. An object is similar to a trait except that (1) its type parameters do not have variance annotations because they are always invariant, and (2) it may have fields, which are initialized when an object

instance is created. When a class does not extend any trait, we omit the empty extends clause $<: \{\}$. When a class has no type parameters, we omit the empty brackets; we write T for $T[]$. A method may have type parameters as well, but they have both lower and upper bounds. We omit clauses when no bounds are specified; for example, we write $T[+P]$ for $T[+P <: \{\}]$ and $m[Q, \{Q\} <: S]$ for $m[\{\} <: Q <: \{\}, \{Q\} <: S <: \{\}]$. By default, each type parameter has an upper bound `Any` and a lower bound `Bottom`, which we describe in the next subsection.

The bounds of a type parameter place restrictions on valid instantiations of the type parameter; a valid type instantiation must be a subtype of all its upper bounds and, at the same time, a supertype of all its lower bounds, if any. We write $+$ for covariance, $-$ for contravariance, and $=$ for invariance. As discussed in Section 2.3, variance annotations of type parameters define subtype relations between different instances of a generic trait. For example, consider the following trait declarations:

```
trait U1 — end
trait U2 <: {U1} — end
trait T[+P, -Q, =S] — end
```

where “—” denotes omitted material. Because U_2 is a subtype of U_1 , the variance annotations of the type parameters of T define subtype relations such as $T[U_2, U_1, U_1] <: T[U_1, U_2, U_1]$.

Expressions and types are conventional. One can refer to a field or bound variable simply by mentioning it; the keyword `self` refers to the target object for the currently executing method. Functions are first-class values; $((\bar{x}:\bar{\tau}): \tau \Rightarrow e)$ is a (typed) function constructor, and we write $e@(\bar{e})$ for function calls. An object creation $O[\bar{\tau}](\bar{e})$ makes an instance of O by instantiating its type parameters with $\bar{\tau}$ and initializing its fields with the values of $\bar{e}$. While tuples are not first-class values in FGFV (they would be trivial to add, but are not needed for our purposes), we use tuple types $(\bar{\tau})$ to denote the domain types of functions and methods, which may have multiple parameters. The type `Any` is a supertype of every type (and `Bottom`, used below, is a subtype of every type but is intentionally not expressible in the surface syntax).

3.3 Well-formedness

We define well-formedness of a program and its declarations as shown in Figure 3. A type-declaration environment Δ is a set of class declarations and type parameter bindings, and a variable-type environment Γ is a sequence of bindable variables and their types. Checking well-formedness may introduce *internal types*:

Internal type $\alpha ::= P \mid C \mid (\bar{\alpha}) \mid (\alpha \rightarrow \alpha) \mid \text{Any} \mid \text{Bottom} \mid (\alpha \sqcup \alpha) \mid (\alpha \sqcap \alpha)$

which contain `Bottom`, union types, and intersection types. Type `Bottom` is a subtype of every type; hence, `Any` and `Bottom` are the top and bottom, respectively, of the type lattice. Metavariables $\alpha, \gamma, \rho, \chi, \eta$ range over internal types. We introduce metavariables K, C, J , and M , analogous to κ, c, t , and μ , respectively, where types τ in the latter are replaced by internal types α in the former.

Rule [T-PROGRAM] states that a program is well-formed if its class declarations have distinct names and are well-formed, and its top-level expression is well-formed. The type of a program is the type of its top-level expression; the metavariable g represents a *ground type*, which is a type that contains no type parameters. We describe well-formedness and type checking of expressions in Section 5.1.

Rules [D-TRAIT] and [D-OBJECT] check well-formedness of class declarations. For the declaration of a class C , they verify that the type parameter names of the declaration are distinct, that the types used in the declaration are well-formed, and that the declared methods are well-formed. In addition, as we explain later in this subsection, they make sure that (1) the *proper ancestors* of C , which are all the traits C extends, by explicitly extending and transitively inheriting as well, are well-formed, and (2) the *visible methods* of C , which are the methods either declared by C or inherited from the traits

Well-formed programs: $\boxed{\vdash \Pi : g}$

$$\frac{\Pi = \bar{\psi}, e \quad \text{distinct}(\overline{\text{name}(\bar{\psi})}) \quad \Delta = \{\bar{\psi}\} \quad \overline{\Delta} \vdash \bar{\psi} \text{ ok} \quad \Delta; \bullet \vdash e : (_, g)}{\vdash \Pi : g} \text{ [T-PROGRAM]}$$

Well-formed trait and object declarations: $\boxed{\Delta \vdash \bar{\psi} \text{ ok}}$

$$\frac{\begin{array}{c} \Delta' = \Delta \cup \{\bar{\xi} : P <: \{\bar{\xi}\}\} \quad \text{distinct}(\bar{P}) \quad \overline{\Delta'} \vdash \bar{\xi} \text{ ok} \quad \overline{\Delta'} \vdash t \text{ ok} \\ \{\bar{J}\} = \text{properAncestors}(\Delta', T[\bar{P}]) \quad \overline{\Delta'; \bar{J} \vdash J_i \text{ and } J_j \text{ ancestors ok}}^{1 \leq i < j \leq \#(\bar{J})} \\ \overline{T \neq \text{name}(\bar{J})} \quad \overline{\Delta'; \text{self}: T[\bar{P}]; \bar{V} \bar{P} \vdash \mu \text{ ok}} \quad \{\bar{d}\} = \text{allVisible}(\Delta', T[\bar{P}]) \\ \overline{\Delta' \vdash d_i \text{ not duplicate of } d_j}^{1 \leq i < j \leq \#(\bar{d})} \quad \overline{\Delta' \vdash d_i \text{ meet } d_j \text{ wrt } \{\bar{d}\} \text{ ok}}^{1 \leq i < j \leq \#(\bar{d})} \\ \overline{\Delta' \vdash d_i \text{ return type wrt } d_j \text{ ok}}^{1 \leq i < j \leq \#(\bar{d})} \quad \overline{\Delta' \vdash d_j \text{ return type wrt } d_i \text{ ok}}^{1 \leq i < j \leq \#(\bar{d})} \end{array}}{\Delta \vdash \text{trait } T[\bar{V} P <: \{\bar{\xi}\}] <: \{\bar{t}\} \bar{\mu} \text{ end ok}} \text{ [D-TRAIT]}$$

$$\frac{\begin{array}{c} \Delta' = \Delta \cup \{\bar{\xi} : P <: \{\bar{\xi}\}\} \quad \text{distinct}(\bar{P}) \quad \overline{\Delta'} \vdash \bar{\xi} \text{ ok} \quad \overline{\Delta'} \vdash t \text{ ok} \\ \{\bar{J}\} = \text{properAncestors}(\Delta', O[\bar{P}]) \quad \overline{\Delta'; \bar{J} \vdash J_i \text{ and } J_j \text{ ancestors ok}}^{1 \leq i < j \leq \#(\bar{J})} \\ \overline{\Delta'; \text{self}: O[\bar{P}], \bar{x}: \bar{\tau}; = \bar{P} \vdash \mu \text{ ok}} \quad \{\bar{d}\} = \text{allVisible}(\Delta', O[\bar{P}]) \\ \overline{\Delta' \vdash d_i \text{ not duplicate of } d_j}^{1 \leq i < j \leq \#(\bar{d})} \quad \overline{\Delta' \vdash d_i \text{ meet } d_j \text{ wrt } \{\bar{d}\} \text{ ok}}^{1 \leq i < j \leq \#(\bar{d})} \\ \overline{\Delta' \vdash d_i \text{ return type wrt } d_j \text{ ok}}^{1 \leq i < j \leq \#(\bar{d})} \quad \overline{\Delta' \vdash d_j \text{ return type wrt } d_i \text{ ok}}^{1 \leq i < j \leq \#(\bar{d})} \\ \overline{\Delta' \vdash \tau \text{ ok}} \quad \text{distinct}(\bar{x}) \end{array}}{\Delta \vdash \text{object } O[\bar{P} <: \{\bar{\xi}\}][\bar{x}: \bar{\tau}] <: \{\bar{t}\} \bar{\mu} \text{ end ok}} \text{ [D-OBJECT]}$$

Well-formed method declarations: $\boxed{\Delta; \Gamma; \bar{V} \bar{P} \vdash \mu \text{ ok}}$

$$\frac{\begin{array}{c} \Delta \vdash [\{\bar{\xi}\} <: Q <: \{\bar{\xi}\}] \text{ ok} \quad \Delta' = \Delta \cup \{\bar{\xi} : Q <: \{\bar{\xi}\}\} \quad \overline{\Delta'} \vdash \bar{\xi} \text{ ok} \quad \overline{\Delta'} \vdash \bar{\xi} \text{ ok} \\ \text{distinct}(\bar{x}) \quad \overline{\Delta' \vdash \tau \text{ ok}} \quad \Delta' \vdash \omega \text{ ok} \quad \Delta'; \Gamma, \bar{x}: \bar{\tau} \vdash e : (_, \rho) \quad \Delta' \vdash \rho <: \omega \\ \text{distinct}(\bar{P}, \bar{Q}) \quad \overline{(V =) \vee (\forall \bar{T}. ((\bar{\tau} \rightarrow \omega) \neq \bar{T}[P]) \vee (\Delta \vdash \bar{T} \text{ variance } V))} \end{array}}{\Delta; \Gamma; \bar{V} \bar{P} \vdash m[\{\bar{\xi}\} <: Q <: \{\bar{\xi}\}][\bar{x}: \bar{\tau}]: \omega = e \text{ ok}} \text{ [D-METHOD]}$$

Fig. 3. Well-formed programs, class declarations, and method declarations

C extends, obey the overloading rules. The only differences are that rule [D-TRAIT] additionally guarantees that a trait does not extend itself, and rule [D-OBJECT] guarantees that the names of the fields are distinct.

Well-formedness of types is conventional. The formal rules are defined in the companion report [Park et al. 2018]. The types Any and Bottom are trivially well-formed. A type parameter P is well-formed under a type-declaration environment Δ , if Δ contains a binding of P . A constructed type $T[\bar{\alpha}]$ or $O[\bar{\alpha}]$ is well-formed, if the trait T or the object O is declared in the given type-declaration environment and all the type arguments $\bar{\alpha}$ are well-formed and satisfy the declared bounds of their corresponding type parameters. Finally, the remaining structural types such as

Well-formed ancestors: $\boxed{\Delta; \bar{J} \vdash J \text{ and } J \text{ ancestors ok}}$

$$\frac{T \neq T'}{\Delta; \bar{J} \vdash T[\bar{\alpha}] \text{ and } T'[\bar{\eta}] \text{ ancestors ok}} \quad [\text{ANC-DIFF-TRAIT}]$$

$$\frac{T[\bar{\gamma}] \in \{\bar{J}\} \quad \Delta \vdash T[\bar{\gamma}] <: T[\bar{\alpha}] \quad \Delta \vdash T[\bar{\gamma}] <: T[\bar{\eta}]}{\Delta; \bar{J} \vdash T[\bar{\alpha}] \text{ and } T[\bar{\eta}] \text{ ancestors ok}} \quad [\text{ANC-SAME-TRAIT}]$$

Well-formed type parameter bindings: $\boxed{\Delta \vdash [\bar{K}] \text{ ok}}$

$$\Delta \vdash [] \text{ ok} \quad [\text{BINDING-EMPTY}]$$

$$\frac{FV(\bar{\chi}') \subseteq \text{parameters}(\Delta) \cup \{\bar{P}\} \quad FV(\bar{\eta}') \subseteq \text{parameters}(\Delta) \quad \Delta \vdash \sqcup \{\bar{\chi}'\} <: \sqcap \{\bar{\eta}'\} \quad \Delta \vdash [\bar{\chi}] <: P <: [\bar{\eta}] \text{ ok}}{\Delta \vdash [\bar{\chi}] <: P <: \{\bar{\eta}\}, \{\bar{\chi}'\} <: P' <: \{\bar{\eta}'\} \text{ ok}} \quad [\text{BINDING-STEP}]$$

Fig. 4. Well-formed ancestors and type parameter bindings

tuple types, arrow types, union types, and intersection types are well-formed, if all their component types are.

The proper ancestors $\{\bar{J}\}$ of a class C are well-formed, if every pair in $\{\bar{J}\}$ satisfies one of the well-formed ancestors rules shown in Figure 4: $\Delta'; \bar{J} \vdash J_i \text{ and } J_j \text{ ancestors ok}^{1 \leq i < j \leq \#(\bar{J})}$. The well-formed ancestors rules ensure that if a class C extends two instances of a trait T , $T[\bar{\alpha}]$ and $T[\bar{\rho}]$, then C also extends some $T[\bar{\gamma}]$ (which may be $T[\bar{\alpha}]$, $T[\bar{\rho}]$, or a third instance) that extends both $T[\bar{\alpha}]$ and $T[\bar{\rho}]$. Therefore, because the rules guarantee that there always exists a single minimal ancestor when a class extends multiple instances of a trait, no ambiguous method calls are possible due to multiple instances of a trait as discussed in Section 5.2.

The visible methods $\{\bar{d}\} = \{\overline{(C, M)}\}$ of a class obey the overloading rules if every pair in $\{\bar{d}\}$ satisfies the overloading rules described in Section 4.1. The set $\{\overline{(C, M)}\}$ contains pairs of visible method declarations $\bar{M}$ and their defining classes $\bar{C}$.

Rule [D-METHOD] checks well-formedness of method declarations. For the method declaration μ , it verifies that the parameter names of the declaration are distinct; that the class type parameters and the method type parameters of the declaration are collectively distinct; that types used in the declaration are well-formed; that the body expression is well-formed; and that the type of the body expression is a subtype of the declared return type. In addition, as we discuss below, it checks that method type parameter bindings are well-formed and that any references to trait type parameters are correctly used according to their variance annotations.

A sequence of method type parameter bindings is well-formed, if (1) every reference to a method type parameter falls within its scope (which extends to the right of its declaration), (2) no reference to a method type parameter appears in any of the upper bounds, and (3) for each type parameter, the union of its lower bounds is a subtype of the intersection of its upper bounds. As we discuss in Section 5.4, restriction (1) makes dynamic dispatch more efficient and restriction (2) makes it consistent. Figure 4 shows the rules for well-formed type parameter bindings.

Trait type parameters are correctly used according to their variance annotations, if covariant and contravariant type parameters appear only at covariant and contravariant positions, respectively. In order to express this restriction, we introduce a *type context* $\mathcal{T}$, which defines positions where a type can appear within another type. Similarly to the conventional evaluation contexts [Wright and

[Felleisen 1994], a type context $\mathcal{T}$ is a type with one of its components replaced by a hole, denoted by “ $\square$ ”. The type $\mathcal{T}[\tau]$ results from placing a type τ in the hole of $\mathcal{T}$:

Type context $\mathcal{T} ::= \square \mid T[\bar{\tau}, \mathcal{T}, \bar{\tau}] \mid O[\bar{\tau}, \mathcal{T}, \bar{\tau}] \mid (\bar{\tau}, \mathcal{T}, \bar{\tau}) \mid (\mathcal{T} \rightarrow \tau) \mid (\tau \rightarrow \mathcal{T})$

Thus, if $\tau = ((P, T[\bar{Q}]) \rightarrow \text{Any})$, then $\tau = \mathcal{T}[P] = \mathcal{T}'[Q]$ where $\mathcal{T} = ((\square, T[\bar{Q}]) \rightarrow \text{Any})$ and $\mathcal{T}' = ((P, T[\square]) \rightarrow \text{Any})$.

Using type contexts, the last premise of rule [D-METHOD] ensures that the class type parameters of a method appear in correct positions. Considering the arrow type representing the method’s type $\tau' = ((\bar{\tau}) \rightarrow \omega)$, an invariant type parameter can appear anywhere in τ' . However, when a covariant or contravariant type parameter P appears in $\tau' = \mathcal{T}[P]$, the *variance of the type context* of P should match the variance V of P : $\Delta \vdash \mathcal{T} \text{ variance } V$. The variance of a type context is defined inductively: (1) the hole is covariant; (2) components of tuple types, return types of arrow types, and covariant type parameters of traits preserve the variance of their enclosing contexts; (3) parameter types of arrow types and contravariant type parameters of traits flip the variance of their enclosing contexts; and (4) type parameters of objects and invariant type parameters of traits are invariant. For example, if $\tau = ((P, T[\bar{Q}]) \rightarrow \text{Any})$ is a method’s type where P and Q are class type parameters and T has a contravariant type parameter, then P is required to be contravariant or invariant and Q is required to be covariant or invariant.

4 OVERLOADING RULES

Now, we formalize the overloading rules informally discussed in Section 2.2 for FGFV.

4.1 Formalization of Overloading Rules

The overloading rules require that every pair in a set of overloaded method declarations satisfies all three rules defined in Figure 5. To determine whether one method declaration is more specific than the other, while the informal rules in Section 2.2 used the subset relation of their applicable sets, which is not practically checkable, we use the subtype relation of their domain types as in Allen et al. [2011]. Note that even though FGFV additionally supports variance annotations of trait type parameters, the overloading rules for FGF remain the same because covariant or contravariant types affect only the well-formedness of method declarations rather than the validity of overloading.

If two method declarations have different names, they vacuously satisfy all the rules ([NO-DUP-TRIV], [MEET-TRIV], and [RETURN-TRIV]). The No Duplicates Rule also checks whether the domain types of two declarations are different; if the domain type of one declaration is not a subtype of the domain type of the other, they satisfy the rule ([NO-DUP-NOT-LESS] and [NO-DUP-NOT-GTR]). The Meet Rule is satisfied if two domain types exclude each other ([MEET-EXCL]), they are in a subtype relation ([MEET-LESS] and [MEET-GTR]), or the domain type of a third declaration of the same name is the intersection type of the domain types of two declarations ([MEET-THIRD]). The Return Type Rule is satisfied if two domain types are not in a subtype relation ([RETURN-NOT-LESS]), or if two arrow types derived from the two declarations are in a subtype relation ([RETURN-TEST]).

4.2 Quantified Types

As proposed by Bourdoncle and Merz [1997], we represent domain types and arrow types of generic method declarations as existential types $\exists$ and universal types $\forall$, respectively, collectively called *quantified types*. For presentation brevity, we omit C from $d = (C, M)$ where a method declaration $M = m[\bar{K}](\bar{x}:\bar{\alpha}) : \rho = e$ is declared in a class C . The domain type of d is $\text{dom}(d) = \exists[\bar{K}](\bar{\alpha})$ and its arrow type is $\text{arrow}(d) = \forall[\bar{K}](\bar{\alpha}) \rightarrow \rho$. We compare domain types to decide whether one generic method declaration is more specific than the other; an existential type $\exists[\bar{K}]\alpha$ is a subtype of another existential type $\exists[\bar{K}']\alpha'$ if there exists a substitution σ for the type parameters bound

[illegible]

Fig. 5. Overloading rules for FGFV

in $\overline{K'}$ that makes $\sigma\alpha'$ a supertype of α . In contrast, we compare arrow types to see whether the return type of a more specific generic method declaration can be a subtype of the return type of a less specific one; a universal type $\forall[\overline{K}]\alpha$ is a subtype of the other universal type $\forall[\overline{K'}]\alpha'$ if there exists a substitution σ for the type parameters bound in $\overline{K}$ that makes $\sigma\alpha$ a subtype of α' . Figure 6 formally defines these “inner” subtyping rules for quantified types. (We assume the conventional α -conversion to rename bound variables when necessary to satisfy premises that require variable names to be distinct.) Finally, the intersection type of two existential types $\exists[\overline{K}]\alpha$ and $\exists[\overline{K'}]\alpha'$ used in [MEET-EXCL] and [MEET-THIRD] is $\exists[\overline{K}, \overline{K'}](\alpha \sqcap \alpha')$, because whenever both $\sigma\alpha$ and $\sigma'\alpha'$ are supertypes of a type τ for some σ and σ' , $(\sigma \circ \sigma')(\alpha \sqcap \alpha')$ is a supertype of τ as well.

While the subtype relation of domain types is a sound approximation of the subset relation of applicable sets, it is not complete; when the applicable set of one declaration is a subset of that of the other, their domain types may not be in a subtype relation. Consider the following:

$$m[P](x : \text{ArrayList}[P]) - \quad m[Q <: T](y : \text{List}[Q]) - \quad m[R <: T](z : \text{ArrayList}[R]) -$$

for some trait T where `ArrayList` is a subtype `List` and both traits are invariant. We call them d_1 , d_2 , and d_3 in order. Even though they are a valid overloading because the applicable set of d_3 is the intersection of those of d_1 and d_2 , the intersection of the two domain types $\Xi_{\sqcap} = \exists[P, Q <: T](\text{ArrayList}[P] \sqcap \text{List}[Q])$ is not a subtype of the domain type of d_3 , which is $\Xi_3 = \exists[R <: T]\text{ArrayList}[R]$. However, because `ArrayList` extends `List` and both are invariant, any subtypes of theirs are exclusive unless they are both $\text{ArrayList}[S]$ for some type S that is a subtype of

Existential inner subtyping: $\boxed{\Delta \vdash \Xi \lesssim \Xi \text{ using } \sigma}$

$$\frac{\begin{array}{l} \Delta' = \Delta \cup \{\{\overline{\chi}\} <: P <: \{\overline{\eta}\}\} \quad \{\overline{P}\} \cap FV(\overline{\chi'}, \overline{\eta'}, \alpha') = \emptyset \quad \sigma = \left[\overline{\gamma}/\overline{Q} \right] \quad \overline{\gamma} \neq \text{Bottom} \\ \Delta' \vdash \sigma \text{ on } \langle \overline{Q} \rangle \text{ obeys } \langle \sqcup \{\overline{\chi'}\} \rangle \text{ and } \langle \sqcap \{\overline{\eta'}\} \rangle \quad \Delta' \vdash \alpha <: \sigma \alpha' \end{array}}{\Delta \vdash \exists [\overline{\chi}] <: P <: \{\overline{\eta}\}] \alpha \lesssim \exists [\overline{\chi'}] <: Q <: \{\overline{\eta'}\}] \alpha' \text{ using } \sigma} \text{ [E-SUB]}$$

Universal inner subtyping: $\boxed{\Delta \vdash \Upsilon \lesssim \Upsilon \text{ using } \sigma}$

$$\frac{\begin{array}{l} \Delta' = \Delta \cup \{\{\overline{\chi'}\} <: Q <: \{\overline{\eta'}\}\} \quad \{\overline{Q}\} \cap FV(\overline{\chi}, \overline{\eta}, \alpha) = \emptyset \quad \sigma = \left[\overline{\gamma}/\overline{P} \right] \quad \overline{\gamma} \neq \text{Bottom} \\ \Delta' \vdash \sigma \text{ on } \langle \overline{P} \rangle \text{ obeys } \langle \sqcup \{\overline{\chi}\} \rangle \text{ and } \langle \sqcap \{\overline{\eta}\} \rangle \quad \Delta' \vdash \sigma \alpha <: \alpha' \end{array}}{\Delta \vdash \forall [\overline{\chi}] <: P <: \{\overline{\eta}\}] \alpha \lesssim \forall [\overline{\chi'}] <: Q <: \{\overline{\eta'}\}] \alpha' \text{ using } \sigma} \text{ [U-SUB]}$$

Existential reduction: $\boxed{\Delta \vdash \Xi \xrightarrow{\Xi} \Xi \text{ using } \sigma}$

$$\frac{\begin{array}{l} \Delta \vdash \alpha \neq \text{Bottom} \Rightarrow C \quad \text{toConstraint}(\{\overline{K}\}) = C' \\ \Delta \vdash \text{unify}(C \wedge C') = (\sigma, C'') \quad \text{toBounds}(C'') = \{\overline{K'}\} \end{array}}{\Delta \vdash \exists [\overline{K}] \alpha \xrightarrow{\Xi} \exists [\overline{K'}] \sigma \alpha \text{ using } \sigma} \quad \frac{\text{otherwise}}{\Delta \vdash \exists [\overline{K}] \alpha \xrightarrow{\Xi} \exists [\overline{K}] \alpha \text{ using } []}$$

Universal reduction: $\boxed{\Delta \vdash \Upsilon \xrightarrow{\Xi} \Upsilon \text{ using } \sigma}$

$$\frac{\Delta \vdash \exists [\overline{K}] \alpha \xrightarrow{\Xi} \exists [\overline{K'}] \alpha' \text{ using } \sigma}{\Delta \vdash \forall [\overline{K}] (\alpha \rightarrow \omega) \xrightarrow{\Xi} \forall [\overline{K'}] (\alpha' \rightarrow \sigma \omega) \text{ using } \sigma}$$

Existential subtyping: $\boxed{\Delta \vdash \Xi \sqsubseteq \Xi \text{ using } \sigma} \quad \boxed{\Delta \vdash \Xi \sqsubseteq \Xi}$

$$\frac{\Delta \vdash \Xi \xrightarrow{\Xi} \Xi' \text{ using } _ \quad \Delta \vdash \Xi' \lesssim \Xi' \text{ using } \sigma}{\Delta \vdash \Xi \sqsubseteq \Xi' \text{ using } \sigma} \quad \frac{\Delta \vdash \Xi \sqsubseteq \Xi' \text{ using } _}{\Delta \vdash \Xi \sqsubseteq \Xi'}$$

Universal subtyping: $\boxed{\Delta \vdash \Upsilon \sqsubseteq \Upsilon \text{ using } \sigma} \quad \boxed{\Delta \vdash \Upsilon \sqsubseteq \Upsilon}$

$$\frac{\Delta \vdash \Upsilon' \xrightarrow{\Xi} \Upsilon'' \text{ using } _ \quad \Delta \vdash \Upsilon \lesssim \Upsilon'' \text{ using } \sigma}{\Delta \vdash \Upsilon \sqsubseteq \Upsilon' \text{ using } \sigma} \quad \frac{\Delta \vdash \Upsilon \sqsubseteq \Upsilon' \text{ using } _}{\Delta \vdash \Upsilon \sqsubseteq \Upsilon'}$$

Fig. 6. Subtype relations of quantified types

T . Thus, $\Xi_{\sqcap}$ denotes the same applicable set as $\exists [S <: T] \text{ArrayList}[S]$, which is a subtype of Ξ_3 using $[R/S]$.

Therefore, in order to reduce the gap between the subset relation of applicable sets and the subtype relation of domain types, we extend the subtype relation of existential types using *existential reduction*. As defined in Figure 6, existential reduction infers a simplified existential type from a given existential type $\exists [\overline{K}] \alpha$. It first collects constraints C that make α different from Bottom: $\Delta \vdash \alpha \neq \text{Bottom} \Rightarrow C$. It then tries to find types that instantiate the type parameters bound in $\overline{K}$ or simply collects their bounds by solving C and the constraints on the type parameters; it makes a substitution σ for the type parameters that have types to instantiate with and constraints C'' for the type parameters that have only bounds. Using the substitution σ and type parameter bindings $\overline{K'}$ generated from C'' , it produces a simplified existential type $\exists [\overline{K'}] \sigma \alpha$. Finally, in the extended subtype relation, one existential type is a subtype of the other if a reduced existential type of the former is a subtype of the latter by the inner subtyping rule as shown in Figure 6.

Collecting constraints C that make a type α different from another type α' requires various relations between types such as subtype, non-subtype, equivalence, non-equivalence, exclusion, and non-exclusion relations. A constraint is inductively defined with boolean literals, true and false, and simple type relations like subtype, non-subtype, exclusion, and non-exclusion relations as base cases, and conjunction and disjunction of constraints as inductive steps. We also defined the various type relations inductively on the structure of types. In addition, we defined all the type relations in a way that supports complete and disjoint case analysis for types. The entire type relations are formally defined in the companion report [Park et al. 2018].

Arrow types are compared only in rule [RETURN-TEST], to see whether the return type of a more specific generic method declaration can be a subtype of the return type of a less specific one. As the informal rule in Section 2.2 specifies, Return Type Rule checks that for two declarations d_1 and d_2 where d_1 is more specific than d_2 , and for any non-Bottom type γ that d_1 is applicable to, the return type of an instance of d_2 that is applicable to γ is a supertype of the return type of some instance of d_1 that is applicable to γ . When the arrow types of d_1 and d_2 are $\forall[\bar{\kappa}](\alpha \rightarrow \rho)$ and $\forall[\bar{\kappa}'](\alpha' \rightarrow \rho')$, respectively, [RETURN-TEST] checks this by checking whether the arrow type of d_1 is a subtype of $\forall[\bar{\kappa}, \bar{\kappa}']((\alpha \sqcap \alpha') \rightarrow \rho')$. Since d_1 is more specific than d_2 and is applicable to γ , there exist substitutions σ_1 and σ_2 that make d_1 and d_2 applicable to γ , respectively, thus both $\sigma_1\alpha$ and $\sigma_2\alpha'$ are supertypes of γ , therefore so is $(\sigma_1 \circ \sigma_2)(\alpha \sqcap \alpha')$. Thus, if the arrow type of d_1 is a subtype of $\forall[\bar{\kappa}, \bar{\kappa}']((\alpha \sqcap \alpha') \rightarrow \rho')$ using some substitution σ , it makes $\sigma\alpha$ a supertype of $(\alpha \sqcap \alpha')$, which makes $(\sigma_1 \circ \sigma_2 \circ \sigma)\alpha$ a supertype of γ , and $\sigma\rho$ a subtype of ρ' . Hence, $\sigma_1 \circ \sigma_2 \circ \sigma$ is the one that makes the instance of d_1 that is applicable to γ and has the return type that is a subtype of the instance of d_2 . In summary, [RETURN-TEST] ensures that for any instance of d_2 instantiated with σ_2 that is applicable to γ , there exists an instance of d_1 instantiated with $\sigma_1 \circ \sigma_2 \circ \sigma$ that is applicable to γ and has a return type that is a subtype of the return type of the instance of d_2 .

In the same manner as for existential types, we extend the subtype relation of universal types using *universal reduction* as defined in Figure 6. When [RETURN-TEST] checks whether one universal type is a subtype of the other, the domain type of the latter is an intersection type. Thus, in order to utilize the extended expressivity by existential reduction, universal reduction takes a universally quantified arrow type $\forall[\bar{K}](\alpha \rightarrow \omega)$, simplifies its domain type to $\exists[\bar{K}']\alpha'$ using σ via existential reduction, and produces a simplified universal type $\forall[\bar{K}'](\alpha' \rightarrow \sigma\omega)$. In the extended subtype relation, one universal type is a subtype of the other if the former is a subtype of a reduced universal type of the latter by the inner subtyping rule as shown in Figure 6.

4.3 Unique Existence of the Most Specific Declaration

THEOREM 4.1. *For any method invocation, there always exists a unique most specific method declaration in its set of applicable method declarations.*

PROOF SKETCH. We first show that the more specific relation of method declarations is a partial order. It is clearly reflexive and transitive since the existential subtype relation is reflexive and transitive. The No Duplicates Rule guarantees that it is antisymmetric. Under the partial order, a set of applicable method declarations is a meet semilattice. The Meet Rule guarantees the existence of a meet for every nonempty subset of the set. Proving the existence of the most specific method declaration is straightforward by structural induction on the meet semilattice. $\square$

5 OVERLOADING RESOLUTION

Method overloading is resolved both at compile time for type checking and at run time for evaluation, and these *static and dynamic overloading resolutions* should be consistent. For a method invocation

Well-formed expressions: $\boxed{\Delta; \Gamma \vdash e : (\epsilon, \alpha)}$

$$\frac{\Delta; \Gamma \vdash e : (\epsilon, C) \quad \frac{\Delta; \Gamma \vdash e' : (\epsilon', \eta) \quad msav(\Delta, (C, m), (C, \bar{\eta})) = m \llbracket (-) : \rho = -}{\Delta; \Gamma \vdash e.m(\bar{e}') : (\epsilon.m(\bar{e}'): \rho, \rho)} \quad [\text{T-METHOD}]$$

Most specific applicable visible method: $\boxed{msav(\Delta, (C, m), \alpha) = D}$

$$\frac{\begin{array}{l} mostSpecific(\Delta, applicable(\Delta, visible(\Delta, (C, m)), \alpha)) = \{\overline{(d, \{\bar{\sigma}\})}\} \quad \#(\overline{(d, \{\bar{\sigma}\})}) = 1 \\ (d', \{\bar{\sigma}'\}) \in \{\overline{(d, \{\bar{\sigma}\})}\} \quad \overline{instantiate}(\underline{d'}, \bar{\sigma}') = D' \quad D \in \{\overline{D'}\} \end{array}}{msav(\Delta, (C, m), \alpha) = D} \quad [\text{MSAV}]$$

Fig. 7. Well-formed method invocation expressions

expression, static overloading resolution chooses an instance of the most specific method declaration that is applicable to the static types of the method arguments; dynamic overloading resolution then must select an instance of the most specific method declaration that is applicable to the dynamic types of the arguments and whose return type is a subtype of the static type of the expression.

5.1 Static Overloading Resolution

As discussed in Section 2.3, in order to make type-sound choices, the static types of method invocation expressions must be available at run time. Thus, checking well-formedness of an expression e produces its type-annotated version ϵ and its type α as shown in Figure 7. A *type-annotated internal expression* is just like an expression except that all the method invocation expressions in it are annotated with their static types:

Type-annotated internal expression $\epsilon ::= z \mid ((\bar{x}:\bar{\alpha}): \alpha \Rightarrow \epsilon) \mid \epsilon @ (\bar{\epsilon}) \mid O \llbracket \bar{\alpha} \rrbracket (\bar{\epsilon}) \mid \epsilon.m(\bar{\epsilon}): \alpha$

Figure 7 shows only [T-METHOD] for method invocation, since the rules for the other kinds of expressions are conventional. The full definitions are available in the companion report [Park et al. 2018]. For a method invocation $e.m(\bar{e}')$, after checking well-formedness of expressions e and $\bar{e}'$, the rule performs static overloading resolution via rule [MSAV] to find an instance of the most specific applicable visible method declaration, and annotates the method invocation with the return type of the chosen instance. Rule [MSAV] first selects the most specific applicable visible method declaration d' with a set of substitutions $\{\bar{\sigma}'\}$ using the static types of the method arguments α . The unique existence of the most specific applicable visible method declaration is guaranteed by Theorem 4.1. Then, the rule selects one instance D out of instantiations of d' with $\{\bar{\sigma}'\}$.

5.2 Dynamic Overloading Resolution

When evaluating a method invocation expression, dynamic overloading resolution selects a method declaration to call by using the dynamic types of the arguments and the static type of the expression. We describe the dynamic overloading resolution in a top-down manner from the dynamic semantics rule for method invocation (Section 5.2.1) to the details of its helper judgments (Sections 5.2.2 and 5.2.3), present a concrete example (Section 5.2.4), and prove its correctness (Section 5.2.5).

5.2.1 Dynamic Semantics.

Dynamic overloading resolution uses additional metavariables as in Figure 8. A class-declaration environment Ψ is a set of class declarations. A ground type g is a dynamically available type, which is an internal type that does not contain type parameters nor `Bottom`; our type system requires that both static and dynamic type inference algorithms instantiate type parameters with non-`Bottom`

Class-declaration environment	$\Psi ::= \{\bar{\psi}\}$
Ground type	$g ::= T[\bar{g}] \mid O[\bar{g}] \mid (\bar{g}) \mid (g \rightarrow g) \mid \text{Any} \mid (g \sqcup g) \mid (g \sqcap g)$
Value	$v ::= O[\bar{g}](\bar{v}) \mid ((\bar{x}:\bar{g}):g \Rightarrow \epsilon)$
Ilk	$k ::= O[\bar{g}] \mid (\bar{g}) \mid (g \rightarrow g)$
Evaluation context	$E ::= \square \mid E\bar{\epsilon}(\bar{\epsilon}) \mid v\bar{\epsilon}(\bar{v}, E, \bar{\epsilon}) \mid O[\bar{g}](\bar{v}, E, \bar{\epsilon}) \mid E.m(\bar{\epsilon}):g \mid v.m(\bar{v}, E, \bar{\epsilon}):g$
Redex	$R ::= v\bar{\epsilon}(\bar{v}) \mid v.m(\bar{v}):g$

Dynamic semantics: $\boxed{\Psi \vdash E[R] \longrightarrow E[\epsilon]}$

$$\frac{v'' = O[\bar{g}](\bar{v}) \quad \text{object } O[\bar{g}](\bar{x}:\bar{g}) \text{ end} \in \Psi \quad \langle \bar{d} \rangle = \text{topologicalSort}(\Psi, \text{visible}(\Psi, (O[\bar{g}], m))) \\ (-, m[\bar{g}](\bar{x}':\bar{g}): \bar{g} \Rightarrow \epsilon) = \text{firstDispatchable}(\Psi, \langle \bar{d} \rangle, (\text{ilk}(v''), \text{ilk}(v'), g'))}{\Psi \vdash E[v''.m(\bar{v}):g'] \longrightarrow E[v''/\text{self}, \bar{v}/x, \bar{v}'/x']\epsilon} \quad [\text{R-METHOD}]$$

First dispatchable method instance: $\boxed{\text{firstDispatchable}(\Psi, \langle \bar{d} \rangle, k, g) = D}$

$$\frac{\langle \bar{d} \rangle = \langle \bar{d}', d'', _ \rangle \quad \neg(\Psi \vdash \bar{k} \text{ and } \bar{g} \text{ dispatch } d' \text{ using } _) \\ \Psi \vdash k \text{ and } g \text{ dispatch } d'' \text{ using } \sigma \quad D = \text{instantiate}(d'', \sigma)}{\text{firstDispatchable}(\Psi, \langle \bar{d} \rangle, k, g) = D} \quad [\text{FIRST-DISPATCHABLE}]$$

Dispatch semipredicate: $\boxed{\Psi \vdash k \text{ and } g \text{ dispatch } d \text{ using } \sigma}$

$$\frac{\text{arrow}(d) = \forall[\bar{K}](\alpha \rightarrow \rho) \quad \Psi \vdash k \text{ match } [\bar{K}] \alpha + \text{bounds } \langle \langle \bar{l} \rangle \rangle \text{ and } \langle \langle \bar{u} \rangle \rangle \\ \Psi \vdash g \text{ match } [\bar{K}] \rho - \text{bounds } \langle \langle \bar{l}' \rangle \rangle \text{ and } \langle \langle \bar{u}' \rangle \rangle \\ \Psi \vdash \text{solve } [\bar{K}] \text{ bounds } \sqcup_{\bar{K}}^{\times} \langle \langle \bar{l} \rangle \rangle, \langle \langle \bar{l}' \rangle \rangle \rangle \text{ and } \prod_{\bar{K}}^{\times} \langle \langle \bar{u} \rangle \rangle, \langle \langle \bar{u}' \rangle \rangle \rangle \text{ using } \sigma}{\Psi \vdash k \text{ and } g \text{ dispatch } d \text{ using } \sigma} \quad [\text{DISPATCH}]$$

Fig. 8. Dynamic semantics of method invocation expressions

types, and that all the type parameters are instantiated with ground types at run time. We call the run-time type of a value the *ilk* of the value; the ilk of an object instance $O[\bar{g}](\bar{v})$ is $O[\bar{g}]$, and the ilk of a function $((\bar{x}:\bar{g}):g \Rightarrow \epsilon)$ is $(\bar{g}) \rightarrow g'$. Similarly to domain types, we write $(\bar{g})$ to denote ilks of multiple function parameters. Evaluation contexts E and redexes R are conventional.

For a method invocation, rule [R-METHOD] seeks the most specific method declaration with a *dispatchable instance* that is applicable to the ilk of the arguments and whose return type is a subtype of the annotated static return type. We say a method declaration is *dispatchable* to the ilk of arguments and the static return type if it has a dispatchable instance. For the sake of efficiency, rule [R-METHOD] first topologically sorts all the visible method declarations in a most-to-least specific order. (This order is statically determinable, so the sorting can be done at compile time.) The rule then checks the *dispatch semipredicate* for each method declaration in the sorted list until it finds the first method declaration d'' dispatchable to the given ilk of the arguments k and the static return type g with a dispatchable instance $D = \text{instantiate}(d'', \sigma)$.

The dispatch semipredicate defined in rule [DISPATCH] checks if a given method declaration d is dispatchable to a given ilk k and a static return type g . The dispatch semipredicate is satisfied using substitution σ if d has a dispatchable instance (obtained by deleting the method type parameters of d and applying σ to its argument types, return type, and body). Metavariable l ranges over lower type bounds, and u ranges over upper type bounds. We write $\langle \bar{l} \rangle$ to denote ordered lists. The dispatch semipredicate consists of three steps: (1) computing type bounds $\langle \langle \bar{l} \rangle \rangle$ and $\langle \langle \bar{u} \rangle \rangle$

Type matching a ground type to an internal type:

$$\boxed{\Psi \vdash g \text{ match } \llbracket \bar{K} \rrbracket \alpha \text{ } V \text{ bounds } \langle \langle \bar{l} \rangle \rangle \text{ and } \langle \langle \bar{u} \rangle \rangle}$$

$$\frac{
\begin{array}{c}
g \text{ is not an intersection type} \quad g \text{ is not a union type} \\
l = (\text{if } (P = \underline{P_i}) \text{ then } \underline{g} \text{ else Bot}) \quad \underline{u} = \text{Any} \quad 1 \leq i \leq \#(\bar{P}) = \#(\bar{u})
\end{array}
}{
\Psi \vdash g \text{ match } \llbracket \{\bar{\chi}\} <: P <: \{\bar{\eta}\} \rrbracket P_i + \text{ bounds } \langle \langle \bar{l} \rangle \rangle \text{ and } \langle \langle \bar{u} \rangle \rangle
} \quad [\text{M-PARAM-CO}]$$

$$\frac{
\Psi \vdash g \text{ match } \llbracket \bar{K} \rrbracket \alpha \text{ } V \text{ bounds } \langle \langle \bar{l} \rangle \rangle \text{ and } \langle \langle \bar{u} \rangle \rangle
}{
\Psi \vdash (\bar{g}) \text{ match } \llbracket \bar{K} \rrbracket (\bar{\alpha}) \text{ } V \text{ bounds } \bigsqcup_{\bar{K}}^{\times} \langle \langle \bar{l} \rangle \rangle \text{ and } \prod_{\bar{K}}^{\times} \langle \langle \bar{u} \rangle \rangle
} \quad [\text{M-TUPLE-TUPLE-SUCCESS}]$$

$$\frac{
\#(\bar{g}) \neq \#(\bar{\alpha})
}{
\Psi \vdash (\bar{g}) \text{ match } - (\bar{\alpha}) - \text{ bounds } \langle \rangle \text{ and } \langle \rangle
} \quad [\text{M-TUPLE-TUPLE-FAILURE}]$$

$$\frac{
\begin{array}{c}
\alpha \text{ is not an intersection type} \quad \alpha \text{ is not a union type} \\
\Psi \vdash g \text{ match } \llbracket \bar{K} \rrbracket \underline{\alpha} - \text{ bounds } \langle \langle \bar{l} \rangle \rangle \text{ and } \langle \langle \bar{u} \rangle \rangle
\end{array}
}{
\Psi \vdash \bigsqcup \langle \bar{g} \rangle \text{ match } \llbracket \bar{K} \rrbracket \alpha - \text{ bounds } \langle \langle \bar{l} \rangle \rangle \text{ and } \langle \langle \bar{u} \rangle \rangle
} \quad [\text{M-UNION-WHATEVER-CONTRA}]$$

Fig. 9. Type matching a ground type to an internal type (selected rules)

that will make the domain type of the method declaration d be a supertype of the ilk k , so that d will be applicable to k , (2) computing type bounds $\langle \langle \bar{l}' \rangle \rangle$ and $\langle \langle \bar{u}' \rangle \rangle$ that make the static return type g a supertype of the return type of d to make sure that the dynamic dispatch preserves type, and (3) solving those type bounds to get a valid substitution σ . (If any of these steps fails, the semipredicate is false: the declaration is not applicable.) The computed type bounds are either ground types or Bot. The dispatch semipredicate computes type bounds by type matching as described in the next section.

5.2.2 Computing Bounds by Type Matching.

The match rules in Figure 9 compute lower and upper type bounds $\langle \langle \bar{l} \rangle \rangle$ and $\langle \langle \bar{u} \rangle \rangle$ for method type parameters in $\bar{K}$ by matching a ground type g against an internal type α according to a variance mark V . When V is respectively $+$, $-$, or $=$, the rules compute the type bounds that make g respectively a subtype, a supertype, or an equivalent type of α . For example, as in [M-PARAM-CO], if a ground type g is matched against a method type parameter P_i according to $+$, g becomes the lower bound of P_i so that g is a subtype of P_i as long as P_i satisfies its type bounds $g <: P_i <: \text{Any}$.

The match rules compute lower and upper type bounds in the form of two lists of lists $\langle \langle \bar{l} \rangle \rangle$ and $\langle \langle \bar{u} \rangle \rangle$ where $\#(\langle \bar{l} \rangle) = \#(\langle \bar{u} \rangle)$ and $\#(\bar{K}) = \#(\bar{l}_i) = \#(\bar{u}_i)$. Matching may produce multiple solutions; each inner list is one possible solution. Multiple solutions come from cases with union types and intersection types as in rule [M-UNION-WHATEVER-CONTRA], which matches a union ground type $\bigsqcup \langle \bar{g} \rangle$ against α according to $-$ and computes type bounds that make $\bigsqcup \langle \bar{g} \rangle$ a supertype of α . Because a union type $\bigsqcup \langle \bar{g} \rangle$ is a supertype of α if any component g_i is a supertype of α , multiple components may be supertypes of α , which may lead to multiple type bounds for each of those components. Since we cannot decide which one of the multiple type bounds is the one eventually satisfying all the constraints, the match rules compute all possible solutions, which are then winnowed in the solving step. To collect all the bounds, the match rules cover failure cases as well so that the rules do not stop collecting bounds even when one component fails type matching.

$$\begin{array}{c}
\text{Solving type bounds: } \boxed{\Psi \vdash \text{solve } \llbracket \bar{K} \rrbracket \text{ bounds } \langle \bar{l} \rangle \text{ and } \langle \bar{u} \rangle \text{ using } \sigma} \\
\quad \Psi \vdash \text{solve } \llbracket \rrbracket \text{ bounds } \langle \rangle \text{ and } \langle \rangle \text{ using } [] \quad \text{[SOLVE-EMPTY]} \\
\\
\frac{\#(\langle \bar{l}, l' \rangle) = \#(\langle \bar{u}, u' \rangle) = n \geq 1 \quad 1 \leq i \leq n \quad g' = (u'_i \sqcap \sqcap \{\bar{\eta}'\}) \quad \Psi \vdash l'_i <: g' \quad \Psi \vdash g' \text{ match } \llbracket \bar{K} \rrbracket \chi' - \text{bounds } \langle \bar{l}'' \rangle \text{ and } \langle \bar{u}'' \rangle}{\Psi \vdash \text{solve } \llbracket \bar{K} \rrbracket \text{ bounds } \sqcup_{\bar{K}}^{\times} \langle \langle \bar{l}'' \rangle, \langle \bar{l}_i \rangle \rangle \text{ and } \sqcap_{\bar{K}}^{\times} \langle \langle \bar{u}'' \rangle, \langle \bar{u}_i \rangle \rangle \text{ using } [\bar{g}/\bar{P}]} \text{[SOLVE-STEP]} \\
\Psi \vdash \text{solve } \llbracket \bar{K}, \{\bar{\chi}'\} <: P' <: \{\bar{\eta}'\} \rrbracket \text{ bounds } \langle \bar{l}, l' \rangle \text{ and } \langle \bar{u}, u' \rangle \text{ using } [\bar{g}/\bar{P}, g'/P']
\end{array}$$

Fig. 10. Solving type bounds

For example, rule [M-TUPLE-TUPLE-FAILURE] specifies a failure case due to size mismatch of tuple types; rather than actually failing, it succeeds but delivers empty lists of solutions.

Lists of solutions for component types are combined in the manner of a Cartesian product. For instance, rule [M-TUPLE-TUPLE-SUCCESS] reduces $\sqcup_{\bar{K}}^{\times} \langle \langle \bar{l}_i \rangle \rangle^i \left(\sqcap_{\bar{K}}^{\times} \langle \langle \bar{u}_i \rangle \rangle^i \right)$ by computing componentwise unions (intersections) for all the combinations of choosing one type bound from each i -th list of the type bounds. Hence, inductively matching n components whose i -th component has m_i type bounds produces a list of $m_1 \times \dots \times m_n$ type bounds (thus if matching any component fails and produces $\langle \rangle$, the whole type matching fails and results in $\langle \rangle$). For example,

$$\begin{aligned}
& \sqcup_{\bar{K}}^{\times} \langle \langle A, B \rangle, \langle C, D \rangle \rangle, \langle \langle E, F \rangle, \langle G, H \rangle, \langle I, J \rangle \rangle = \\
& \langle \langle A \sqcup E, B \sqcup F \rangle, \langle A \sqcup G, B \sqcup H \rangle, \langle A \sqcup I, B \sqcup J \rangle, \langle C \sqcup E, D \sqcup F \rangle, \langle C \sqcup G, D \sqcup H \rangle, \langle C \sqcup I, D \sqcup J \rangle \rangle.
\end{aligned}$$

5.2.3 Solving the Bounds.

The last step of the dispatch semipredicate is to solve the computed type bounds for method type parameters as in Figure 10. The solve rules compute a valid substitution σ for all the method type parameters in $\bar{K}$, which satisfies the declared bounds in $\bar{K}$ and the computed bounds from type matching $\langle \bar{l} \rangle$ and $\langle \bar{u} \rangle$. The solving step succeeds if any of the type bounds leads to a valid substitution. For declared type bounds for method type parameters $\bar{P}, P'$ and their computed type bounds $\langle \bar{l}, l' \rangle$ and $\langle \bar{u}, u' \rangle$ from type matching, rule [SOLVE-STEP] nondeterministically selects i -th type bounds $\langle \bar{l}_i, l'_i \rangle$ and $\langle \bar{u}_i, u'_i \rangle$ that successfully compute a valid substitution. Starting from the rightmost type parameter P' , the rule computes a substitution $[g'/P']$ by taking the intersection of the upper bounds of P' . Because g' is a ground type, the solving step never instantiates method type parameters with **Bottom**. The match premises propagate the choice of g' for P' to the declared lower bounds of P' to collect additional constraints on $\bar{P}$. Then, the rule combines the additional constraints and the originally computed bounds, and inductively solves the combined bounds to compute the substitution $[\bar{g}/\bar{P}]$. From this, it produces a valid substitution $[\bar{g}/\bar{P}, g'/P']$.

Note that the rule selects i -th type bounds nondeterministically while it checks type parameters deterministically from the rightmost one. Because the scope of a well-formed method type parameter extends to the right and method type parameters can appear only in lower bounds, for well-formed type parameters $\{\bar{\chi}\} <: P <: \{\bar{\eta}\}$, the upper bounds $\{\bar{\eta}\}$ do not contain any of $\bar{P}$, and P_j does not appear in the lower bounds of P_i for $i \leq j$. Thus, the choice of g_i for P_i does not affect the upper bounds of P_j , which allows us to compute a substitution in the right-to-left order by taking intersections of upper bounds. Since the choice of g_j for P_j may place additional constraints on P_i 's

that appear in the lower bounds $\{\overline{\chi_j}\}$ of P_j , [SOLVE-STEP] propagates the choice of g_j for P_j to the lower bounds and computes additional constraints, to ensure that g_j is a supertype of all $\{\overline{\chi_j}\}$.

5.2.4 Example.

Let us revisit the dynamic overloading resolution process with a concrete example. Suppose we have two covariant traits `List` and `SortedList` such that $\text{SortedList}[P] <: \text{List}[P]$, two objects `Cons` and `SortedListCons` such that $\text{Cons}[P] <: \text{List}[P]$ and $\text{SortedListCons}[P] <: \text{SortedList}[P]$, and a method `append` that concatenates two lists. Consider the following method declarations:

$$\begin{aligned} \text{append}[P, \{P\} <: Q](x: \text{SortedList}[P], y: \text{SortedList}[Q]): \text{SortedList}[Q] &= _ \\ \text{append}[P, \{P\} <: Q](x: \text{List}[P], y: \text{List}[Q]): \text{List}[Q] &= _ \end{aligned}$$

and call them d_1 and d_2 in order. This is a valid overloading since it satisfies the overloading rules with d_1 being more specific than d_2 . Suppose we have three monomorphic classes C_1 , C_2 , and C_3 such that $C_1 <: C_3$ and $C_2 <: C_3$, and consider the following method invocation expression:

$$\text{append}(l_1, l_2)$$

with two lists l_1 and l_2 whose static types are both $\text{List}[C_3]$ and whose ilks are $\text{SortedListCons}[C_1]$ and $\text{Cons}[C_2]$, respectively. Then, type checking this expression would produce the internal expression $\text{append}(l_1, l_2): \text{List}[C_3]$.

Now, let us see how this method call is dynamically dispatched. Given the ilk of the argument tuple $(\text{SortedListCons}[C_1], \text{Cons}[C_2])$, the static return type $\text{List}[C_3]$, and the topologically sorted list of method declarations $\langle d_1, d_2 \rangle$, dynamic overloading resolution performs the dispatch semipredicate starting from the first declaration d_1 in the list. The dispatch semipredicate on d_1 would succeed if it computed a valid substitution. However, it fails because its first type matching step fails, producing an empty list of solutions:

$$\begin{array}{c} \Psi \vdash \text{SortedListCons}[C_1] \text{ match } [\overline{K}] \text{SortedList}[P] + \text{bounds } \langle \langle C_1, \text{Bottom} \rangle \rangle \text{ and } \langle \langle \text{Any}, \text{Any} \rangle \rangle \\ \Psi \vdash \text{Cons}[C_2] \text{ match } [\overline{K}] \text{SortedList}[Q] + \text{bounds } \langle \rangle \text{ and } \langle \rangle \\ \hline \Psi \vdash (\text{SortedListCons}[C_1], \text{Cons}[C_2]) \text{ match } [\overline{K}] (\text{SortedList}[P], \text{SortedList}[Q]) + \text{bounds } \langle \rangle \text{ and } \langle \rangle \end{array}$$

where $\overline{K} = \{\text{Bottom}\} <: P <: \{\text{Any}\}, \{P\} <: Q <: \{\text{Any}\}$. While the first component matching succeeds, the second component matching fails since $\text{Cons}[C_2]$ is not a subtype of $\text{SortedList}[Q]$. Although matching the static return type against the return type of d_1 succeeds, the combined type bounds for the solving step are the empty lists, which leads to the failure of the dispatch semipredicate:

$$\begin{array}{c} \text{arrow}(d_1) = \forall [\overline{K}] ((\text{SortedList}[P], \text{SortedList}[Q]) \rightarrow \text{SortedList}[Q]) \\ \Psi \vdash (\text{SortedListCons}[C_1], \text{Cons}[C_2]) \text{ match } [\overline{K}] (\text{SortedList}[P], \text{SortedList}[Q]) + \text{bounds } \langle \rangle \text{ and } \langle \rangle \\ \Psi \vdash \text{List}[C_3] \text{ match } [\overline{K}] \text{SortedList}[Q] - \text{bounds } \langle \langle \text{Bottom}, \text{Bottom} \rangle \rangle \text{ and } \langle \langle \text{Any}, C_3 \rangle \rangle \\ \Psi \vdash \text{solve } [\overline{K}] \text{bounds } \langle \rangle \text{ and } \langle \rangle \text{ using } \times \\ \hline \Psi \vdash (\text{SortedListCons}[C_1], \text{Cons}[C_2]) \text{ and } \text{List}[C_3] \text{ dispatch } d_1 \text{ using } \times \end{array}$$

Dynamic overloading resolution then moves on to the next method declaration in the list, d_2 , and performs the dispatch semipredicate on it. In this case, both type matching steps succeed, and the overall result of the dispatch semipredicate depends on the result of the solving step:

We start solving the type bounds from the rightmost type parameter Q . Since we have a single pair of type bounds, that is, $\#(\langle C_1, C_2 \rangle) = \#(\langle \text{Any}, C_3 \rangle) = 1$, there is no need for a nondeterministic choice in rule [SOLVE-STEP]. The substitution for Q is the intersection of the computed upper bound and the declared upper bound: $C_3 \sqcap \text{Any} = C_3$. This choice of substitution C_3 for Q has to be propagated by matching it against the declared lower bound of Q , which is P , to make sure that C_3

$$\begin{array}{c}
\text{arrow}(d_2) = \forall \bar{K} ((\text{List}[P], \text{List}[Q]) \rightarrow \text{List}[Q]) \\
\Psi \vdash (\text{SortedCons}[C_1], \text{Cons}[C_2]) \text{ match } \bar{K} (\text{List}[P], \text{List}[Q]) + \text{bounds } \langle \langle C_1, C_2 \rangle \rangle \text{ and } \langle \langle \text{Any}, \text{Any} \rangle \rangle \\
\Psi \vdash \text{List}[C_3] \text{ match } \bar{K} \text{List}[Q] - \text{bounds } \langle \langle \text{Bottom}, \text{Bottom} \rangle \rangle \text{ and } \langle \langle \text{Any}, C_3 \rangle \rangle \\
\Psi \vdash \text{solve } \bar{K} \text{ bounds } \langle \langle C_1, C_2 \rangle \rangle \text{ and } \langle \langle \text{Any}, C_3 \rangle \rangle \text{ using } \sigma \\
\hline
\Psi \vdash (\text{SortedCons}[C_1], \text{Cons}[C_2]) \text{ and } \text{List}[C_3] \text{ dispatch } d_2 \text{ using } \sigma
\end{array}$$

is a supertype of P . We then combine the type bounds obtained from propagation and the originally given type bounds, and inductively solve the combined bounds to get the substitution for P , which results in $[C_3/P]$ by again taking the intersection of the upper bounds: $C_3 \sqcap \text{Any} = C_3$. Thus, the solving step succeeds with a valid substitution $[C_3/P, C_3/Q]$:

$$\begin{array}{c}
\Psi \vdash C_2 <: C_3 \quad \Psi \vdash C_3 \text{ match } \llbracket \text{Bottom} <: P <: \text{Any} \rrbracket P - \text{bounds } \langle \langle \text{Bottom} \rangle \rangle \text{ and } \langle \langle C_3 \rangle \rangle \\
\Psi \vdash \text{solve } \llbracket \text{Bottom} <: P <: \text{Any} \rrbracket \text{ bounds } \langle \langle C_1 \rangle \rangle \text{ and } \langle \langle C_3 \rangle \rangle \text{ using } [C_3/P] \\
\hline
\Psi \vdash \text{solve } \llbracket \text{Bottom} <: P <: \text{Any}, P <: Q <: \text{Any} \rrbracket \text{ bounds } \langle \langle C_1, C_2 \rangle \rangle \text{ and } \langle \langle \text{Any}, C_3 \rangle \rangle \text{ using } [C_3/P, C_3/Q]
\end{array}$$

As a result, the dispatch semipredicate on d_2 succeeds with $[C_3/P, C_3/Q]$, and the method call expression is dispatched to the following instance of d_2 :

$$D = \text{instantiate}(d_2, [C_3/P, C_3/Q]) = \text{append}(x: \text{List}[C_3], y: \text{List}[C_3]): \text{List}[C_3] = _$$

It shows that the dynamic overloading resolution selects an instance of the most specific applicable method declaration without any ambiguity, and the return type of the chosen instance is a subtype of the static return type, which means that the dynamic overloading resolution preserves type.

5.2.5 Correctness of Dispatch Semipredicate.

We prove the soundness and completeness of the dispatch semipredicate by showing that each of its steps is sound and complete. While we state the lemmas and theorems in prose in this paper, we state them formally in the companion report [Park et al. 2018].

LEMMA 5.1. (Soundness of Type Matching) *The match rules compute sound bounds for method type parameters by matching a ground type g against an internal type α according to a variance mark V . That is, any substitution σ satisfying the bounds makes $\sigma\alpha$ a supertype, a subtype, or an equivalent type of g when V is respectively $+$, $-$, or $=$.*

LEMMA 5.2. (Completeness of Type Matching) *Assume that an internal type α does not contain Bottom, and if α contains any union type or an intersection type γ , γ does not contain any type parameter. For a ground type g and a variance mark V , if a substitution σ on $\langle \bar{P} \rangle$ makes $\sigma\alpha$ a supertype, a subtype, or an equivalent type of g when V is respectively $+$, $-$, or $=$, then the match rules find bounds of $\langle \bar{P} \rangle$ that σ satisfies.*

A subtle point in Lemma 5.2 is the assumptions on type α . While the syntax of the match rules allows any internal types, each match rule is used only with α coming from method declarations. The explicitly annotated types in method declarations are external types, which do not contain Bottom, union types, nor intersection types. On the contrary, inferred types for method type parameters may contain such internal types. For instance, in the following example (in which type annotations appear for the two method calls):

```

object O' [Q]   m'(x: Q): Q = _           end
object O       m[P](x: P, y: P): P = O'[P].m'(x): P   end
O.m(3, true): (Int  $\sqcup$  Boolean)

```

the method type parameter P of m is inferred as $(\text{Int} \sqcup \text{Boolean})$ by the dispatch semipredicate at run time. Since this inferred type then flows into the method declaration of m' , the ilk and the static type of method invocation expression of m' are matched against the union type. However, because

our type system requires that both static and dynamic type inference algorithms instantiate type parameters with non-Bottom types, α does not contain Bottom. In addition, when α contains union types or intersection types, all the type parameters in them are already instantiated with ground types.

LEMMA 5.3. (*Soundness of Solving Bounds*) *The solve rules compute a valid substitution that satisfies both the declared bounds and computed bounds of method type parameters.*

LEMMA 5.4. (*Completeness of Solving Bounds*) *If a valid substitution for method type parameters exists, then the solve rules find one.*

THEOREM 5.5. (*Soundness of Dispatch Semipredicate*) *If the dispatch semipredicate on a method declaration d for an ilk k and a static return type g succeeds, then d is dispatchable to k and g .*

THEOREM 5.6. (*Completeness of Dispatch Semipredicate*) *If a method declaration d is dispatchable to an ilk k and a static return type g , then the dispatch semipredicate on d for k and g succeeds.*

The theorems are proved directly from the four lemmas, which are proved by induction on match rules, the structure of g and a , solve rules, and the number of method type parameters, respectively.

5.3 Type Soundness of FGFV

We now prove the type soundness of FGFV. First, we show that application of a valid substitution to both sides of a subtype relation preserves the relation.

LEMMA 5.7. (*Substitution Preserves Subtype Relation*) *For $\Delta' = \{\{\bar{\chi}\} <: P <: \{\bar{\eta}\}\}$ and $\sigma = [\bar{\gamma}/\bar{P}]$ where $\bar{\gamma}$ satisfy the bounds of their corresponding $\bar{P}$, and free variables of $\bar{\gamma}$ do not include any of $\bar{P}$, if $\Delta \cup \Delta' \vdash \alpha <: \alpha'$ then $\Delta \vdash \sigma\alpha <: \sigma\alpha'$.*

PROOF SKETCH. By structural induction on α and α' using subtype rules. $\square$

Next, we show that no matter which applicable method instance we choose for a method invocation at compile time, there exists a type-safe most specific method instance at run time.

LEMMA 5.8. (*Existence of a Type-safe Most Specific Method Instance*) *For a method invocation expression $e.m(\bar{e}')$ and a type-declaration environment $\Delta = \Psi \cup \{\{\bar{\chi}\} <: P <: \{\bar{\eta}\}\}$, let the static types of e and $(\bar{e}')$ be C_2 and γ respectively, their run-time types be C_1 and k respectively, with $\Psi \vdash C_1 <: \sigma C_2$ and $\Psi \vdash k <: \sigma\gamma$ for some σ that satisfies the bounds of $\bar{P}$, a statically applicable method declaration $(d_2, \{\bar{\sigma}_2\}) \in \text{applicable}(\Delta, \text{visible}(\Delta, (C_2, m)), \gamma)$, and the dynamically most specific applicable method declaration $(d_1, \{\bar{\sigma}_1\}) \in \text{mostSpecific}(\Psi, \text{applicable}(\Psi, \text{visible}(\Psi, (C_1, m)), k))$. If $D_2 \in \{\text{instantiate}(\underline{d}_2, \underline{\sigma}_2)\}$ with $\text{arrow}(D_2) = \forall \llbracket _ \rrbracket (- \rightarrow \rho)$, then $\exists D_1 \in \{\text{instantiate}(\underline{d}_1, \underline{\sigma}_1)\}$ with $\text{arrow}(D_1) = \forall \llbracket _ \rrbracket (- \rightarrow g)$ such that $\Psi \vdash g <: \sigma\rho$.*

PROOF SKETCH. Let $\text{arrow}(d_1) = \forall \llbracket \bar{K}_1 \rrbracket (\alpha_1 \rightarrow \rho_1)$ and $\text{arrow}(d_2) = \forall \llbracket \bar{K}_2 \rrbracket (\alpha_2 \rightarrow \rho_2)$. Because $\Psi \vdash C_1 <: \sigma C_2$, $\Psi \vdash k <: \sigma\gamma$, and d_2 is visible from C_2 and is applicable to γ , by Lemma 5.7, σd_2 is visible from C_1 and is applicable to k , that is $(\sigma d_2, \{\bar{\sigma}'_2\}) \in \text{applicable}(\Psi, \text{visible}(\Psi, (C_1, m)), k)$ with $\{\bar{\sigma}'_2\} \subseteq \{\bar{\sigma}'_2\}$. Thus, $\Psi \vdash \text{dom}(d_1) \sqsubseteq \text{dom}(\sigma d_2)$ and by [RETURN-TEST],

$$\Psi \vdash \forall \llbracket \bar{K}_1 \rrbracket (\alpha_1 \rightarrow \rho_1) \sqsubseteq \forall \llbracket \bar{K}_1, \bar{K}_2 \rrbracket ((\alpha_1 \sqcap \sigma\alpha_2) \rightarrow \sigma\rho_2) \text{ using } \sigma'$$

which implies

$$\Psi \vdash \forall \llbracket \bar{K}_1, \bar{K}_2 \rrbracket ((\alpha_1 \sqcap \sigma\alpha_2) \rightarrow \sigma\rho_2) \xrightarrow{\equiv} \forall \llbracket \bar{K}_3 \rrbracket (\alpha_3 \rightarrow \rho_3) \text{ using } \sigma_r \quad (1)$$

$$\Psi \vdash \forall \llbracket \bar{K}_1 \rrbracket (\alpha_1 \rightarrow \rho_1) \lesssim \forall \llbracket \bar{K}_3 \rrbracket (\alpha_3 \rightarrow \rho_3) \text{ using } \sigma' \quad (2)$$

If $D_2 \in \{\overline{\text{instantiate}(\underline{d}_2, \underline{\sigma}_2)}\}$, then $\text{arrow}(D_2) = \forall \llbracket (\sigma_2 \alpha_2 \rightarrow \sigma_2 \rho_2) \rrbracket$ for some $\sigma_2 \in \{\overline{\sigma}_2\}$ and we have $\Delta \vdash \gamma <: \sigma_2 \alpha_2$, which implies $\Psi \vdash k <: (\sigma \circ \sigma_2) \alpha_2$ by Lemma 5.7. Because $\Psi \vdash k <: \sigma_1 \alpha_1$ for any $\sigma_1 \in \{\overline{\sigma}_1\}$, $(\sigma_1 \circ \sigma_2)(\alpha_1 \sqcap \sigma \alpha_2)$ is not Bottom. Since σ_r represents the minimum requirement that prevents $\sigma_r(\alpha_1 \sqcap \sigma \alpha_2)$ from being Bottom, there exists σ'' such that $\sigma_1 \circ \sigma_2 = \sigma'' \circ \sigma_r$. From (1), we have $\Psi \cup \{\overline{K}_3\} \vdash \sigma_r(\alpha_1 \sqcap \sigma \alpha_2) \equiv \alpha_3$ and $\Psi \cup \{\overline{K}_3\} \vdash (\sigma_r \circ \sigma) \rho_2 \equiv \rho_3$, and from (2), we have $\Psi \cup \{\overline{K}_3\} \vdash \alpha_3 <: \sigma' \alpha_1$ and $\Psi \cup \{\overline{K}_3\} \vdash \sigma' \rho_1 <: \rho_3$. Thus, by Lemma 5.7, $\Psi \vdash k <: (\sigma'' \circ \sigma') \alpha_1$ and $\Psi \vdash (\sigma'' \circ \sigma') \rho_1 <: \sigma(\sigma_2 \rho_2)$ since $\sigma_1 \rho_2 = \rho_2$. Therefore, we have $\sigma'' \circ \sigma' \in \{\overline{\sigma}_1\}$ and there exists $D_1 = \text{instantiate}(\underline{d}_1, \sigma'' \circ \sigma')$ with $\text{arrow}(D_1) = \forall \llbracket (- \rightarrow (\sigma'' \circ \sigma') \rho_1) \rrbracket$ such that $\Psi \vdash (\sigma'' \circ \sigma') \rho_1 <: \sigma(\sigma_2 \rho_2)$. $\square$

We now show that if a type-safe most specific method instance exists, the *firstDispatchable* rule finds one. This lemma follows from the soundness and completeness of the dispatch semipredicate.

LEMMA 5.9. (*Validity of the firstDispatchable Rule*) For a class-declaration environment Ψ , a ground constructed type C , and a method m , let $\{\overline{d'}\} = \text{visible}(\Psi, (C, m))$. For an ilk k and a ground type g , if $(d, \{\overline{\sigma}\}) \in \text{mostSpecific}(\Psi, \text{applicable}(\Psi, \{\overline{d'}\}, k))$ and $D_0 \in \{\overline{\text{instantiate}(\underline{d}, \underline{\sigma})}\}$ with $\text{arrow}(D_0) = \forall \llbracket (- \rightarrow g_0) \rrbracket$ where $\Psi \vdash g_0 <: g$, then $\text{firstDispatchable}(\Psi, \text{topologicalSort}(\Psi, \{\overline{d'}\}), k, g) = D$ with $\text{arrow}(D) = \forall \llbracket (g_1 \rightarrow g_2) \rrbracket$ where $\Psi \vdash k <: g_1$ and $\Psi \vdash g_2 <: g$.

PROOF SKETCH. Let $\text{arrow}(d) = \forall \llbracket \overline{K} \rrbracket (\alpha \rightarrow \rho)$. If $D_0 = \forall \llbracket (\sigma \alpha \rightarrow \sigma \rho) \rrbracket$ for some $\sigma \in \{\overline{\sigma}\}$ with $\Psi \vdash \sigma \rho <: g$, since D_0 is applicable to k , $\Psi \vdash k <: \sigma \alpha$. Therefore, by Theorem 5.6, we have $\Psi \vdash k$ and g dispatch d using σ' for some σ' . Since d is the most specific method declaration that is applicable to k , rule *firstDispatchable* produces D , an instance of d instantiated with σ' : $\text{arrow}(D) = \forall \llbracket (\sigma' \alpha \rightarrow \sigma' \rho) \rrbracket$. By Theorem 5.5, $\Psi \vdash k <: \sigma' \alpha$ and $\Psi \vdash \sigma' \rho <: g$. $\square$

Finally, we prove the type soundness of FGFV. Since the main focus of this paper is resolution of overloaded method invocation and the other cases are conventional, we prove only the method invocation case: a well-typed method invocation can be reduced via [R-METHOD] and the reduced expression preserves the type of the method invocation.

THEOREM 5.10. (*Type Soundness of Method Invocation Expression*) For a class-declaration environment Ψ of a well-formed program and a method invocation expression $O \llbracket \overline{g''} \rrbracket (\overline{v}).m(\overline{v'})$, let $\langle \overline{d} \rangle = \text{topologicalSort}(\Psi, \text{visible}(\Psi, (O \llbracket \overline{g''} \rrbracket, m)))$ for object $O \llbracket _ \rrbracket (x: _)$ — $\text{end} \in \Psi$. If the method invocation is well-typed:

$$\Psi; \bullet \vdash O \llbracket \overline{g''} \rrbracket (\overline{v}).m(\overline{v'}) : (-, g),$$

then $\text{firstDispatchable}(\Psi, \langle \overline{d} \rangle, (O \llbracket \overline{g''} \rrbracket, \overline{\text{ilk}(\overline{v'})}), g) = (-, m \llbracket (x': _) : _ = \epsilon \rrbracket)$ and the body expression of the selected method declaration is successfully evaluated while preserving type:

$$\Psi; \bullet \vdash [O \llbracket \overline{g''} \rrbracket (\overline{v}) / \text{self}] [\overline{v/x}] [\overline{v'/x'}] \epsilon : (-, g') \text{ with } \Psi \vdash g' <: g.$$

PROOF SKETCH.

Let $(d, \{\overline{\sigma}\}) \in \text{mostSpecific}(\Psi, \text{applicable}(\Psi, \text{visible}(\Psi, (O \llbracket \overline{g''} \rrbracket, m)), (O \llbracket \overline{g''} \rrbracket, \overline{\text{ilk}(\overline{v'})})))$. Since $\Psi; \bullet \vdash O \llbracket \overline{g''} \rrbracket (\overline{v}).m(\overline{v'}) : (-, g)$, by Lemma 5.8, there exists $D_1 \in \{\overline{\text{instantiate}(\underline{d}, \underline{\sigma})}\}$ with $\text{arrow}(D_1) = \forall \llbracket (- \rightarrow g_1) \rrbracket$ such that $\Psi \vdash g_1 <: g$. Thus, by Lemma 5.9,

$$D = (-, m \llbracket (x': _) : _ = \epsilon \rrbracket) = \text{firstDispatchable}(\Psi, \langle \overline{d} \rangle, (O \llbracket \overline{g''} \rrbracket, \overline{\text{ilk}(\overline{v'})}), g)$$

with $\text{arrow}(D) = \forall \llbracket (g_1 \rightarrow g_2) \rrbracket$ where $\Psi \vdash (O \llbracket \overline{g''} \rrbracket, \overline{\text{ilk}(\overline{v'})}) <: g_1$ and $\Psi \vdash g_2 <: g$. Since substitution preserves type, when $\Psi; \bullet \vdash [O \llbracket \overline{g''} \rrbracket (\overline{v}) / \text{self}] [\overline{v/x}] [\overline{v'/x'}] \epsilon : (-, g')$, $\Psi \vdash g' <: g_2$, thus $\Psi \vdash g' <: g$. $\square$

THEOREM 5.11. (*Type Soundness of FGFV*) *For a class-declaration environment Ψ of a well-formed program, if $\Psi; \bullet \vdash e : (\epsilon, g)$ and $\Psi \vdash \epsilon \longrightarrow^* v$, then $\Psi; \bullet \vdash v : (_, g')$ and $\Psi \vdash g' <: g$.*

5.4 Discussion

Defining a provable system for an object-oriented language requires addressing various technical challenges due to the structural asymmetries inherent in the language design. One can have a variable of type `Any`, but one cannot have a variable of type `Bottom`. The ilk of an actual argument must be a subtype, not a supertype, of the method's parameter type. The ilk of a returned value must be a subtype, not a supertype, of the method's return type. A trait may declare ancestors but not descendants. Such asymmetries prohibit seemingly symmetric features from being treated as duals. One might think at first glance that upper bounds and lower bounds of type parameters should be related symmetrically (perhaps treated as duals), but they cannot be.

For example, the “Ancestor Meet Rule” (expressed in this paper by rule [ANC-SAME-TRAIT]) was introduced into Fortress in 2012 by Guy Steele and David Chase [Steele and Chase 2012] to design an expressive type system with union and intersection types. This rule permits proofs that (a) if $T[P]$ is covariant in P then $T[(\alpha \sqcap \alpha')] \equiv (T[\alpha] \sqcap T[\alpha'])$, and (b) if $T[P]$ is contravariant in P then $T[(\alpha \sqcup \alpha')] \equiv (T[\alpha] \sqcup T[\alpha'])$ (that last set operator is indeed $\sqcap$, not $\sqcup$). These equivalences allow simplification of type expressions by moving type union and type intersection operators out of type arguments. However, the “obvious other two cases” are not available. When $T[P]$ is covariant in P , $T[(\alpha \sqcup \alpha')]$ is not equivalent to some simple combination of $T[\alpha]$ and $T[\alpha']$; the best one can say is that $(T[\alpha] \sqcup T[\alpha']) <: T[(\alpha \sqcup \alpha')]$.

In 2012, the developers of Fortress explored an alternate set of language restrictions that exploited these asymmetries to ensure that exponential blowup could occur at run time only when union types interacted with contravariance (the dual case of intersection types interacting with covariance could not occur). They then contemplated two alternatives for avoiding the exponential blowup. One possibility is to eliminate contravariant type parameters and contravariant type relationships. This has the consequence that arrow types must be treated as invariant, rather than contravariant, in the type to the left of the arrow. This could make it inconvenient to use higher-order functions such as `map`, but one can compensate by introducing rules of coercion, so that if a function of type $(\alpha \rightarrow \rho)$ is used in a situation where a function of type $(\alpha' \rightarrow \rho')$ is required, it can be coerced to a new function object if $\alpha' <: \alpha$ and $\rho <: \rho'$. The other possibility is to eliminate union types from the type theory; this might require restricting the language so that, for example, in an if-then-else expression, the type of the then subexpression must be either a supertype or a subtype of the else subexpression. Alternatively, some sort of pseudo-join (least common existing supertype) operation could be used. One could go even further, also eliminating intersection types from the type theory by further restricting the language and possibly using a pseudo-meet operation.

We carefully designed our dynamic dispatch algorithm to make a balanced tradeoff between static expressivity and dynamic efficiency. We accept the potential for exponential blowup because we believe that such cases are unlikely to arise in practice and the use of union and intersection types in the type theory, even if not directly expressible by the programmer, makes the language more expressive. Also, while one could in principle allow any lower bound to refer to any type parameter, dynamic overloading resolution would then have to solve a general set of type inequalities, requiring a complex technique with multiple iterations; we chose instead to impose left-to-right scoping on method type parameters, allowing use of a simple, one-pass right-to-left scan. We believe this tradeoff represents a relatively small sacrifice of expressivity for a large gain in dynamic efficiency.

Dynamically inferring type instantiations can incur run-time overhead. In simple invariant cases, we expect the run-time cost to be relatively low but nonzero. For example, if we pass a value of type `List[String]` to a method such as `foo[T](x:List[T])`, T is matched only once and no union

or intersection types are involved. If `List` is invariant in T , it will come down to extracting the type `String` from the type `List[String]` and binding T to that. For more costly covariant and contravariant cases, as in certain JVM implementations, such techniques as caching or dynamic recompilation in reaction to the relative frequencies of types actually encountered may help.

The biggest challenge we faced in trying to find a provable system was grappling with the language asymmetries described above and trying to understand their consequences. In particular, the Fortress team had assumed, because it seemed from several working examples to be in accord with intuition, that once the tightest possible upper and lower bounds for a method type parameter have been found, then the union (or pseudo-union) of the lower bounds should be chosen as the type to be used for that type parameter. We considered this and the hybrid technique of using the intersection of upper bounds only when the union of lower bounds is `Bottom`, but we were unable to complete the proofs of correctness of both the techniques. Since the minimum requirement for a type to instantiate a method type parameter is that it satisfies all the lower and upper bounds of the type parameter, the dynamic dispatch algorithm presented here always chooses the intersection of the upper bounds.

In the work presented here, as in previous work, one of the most difficult parts is finding the necessary substitution σ to satisfy the rule [E-SUB] for existential subtyping during static analysis. This is more difficult than the corresponding calculation during dynamic dispatch because at run time one always matches an ilk (which contains no type parameters) to a type, but at compile time one must match a type against a type, where each type may contain type parameters. Indeed, we do not present a specific static type inference algorithm that finds the substitution σ in rule [E-SUB], but only refer to previous work in this area. However, we conjecture that the match rule presented here can serve this role after being extended to handle the case of matching two type parameters.

6 RELATED WORK

This paper extends the work of [Allen et al. \[2011\]](#) with variance for polymorphic types and a dynamic dispatch algorithm. Allen et al. proposed rules for defining overloaded functions that ensure type safety under symmetric multiple dispatch in an object-oriented language with multiple inheritance, and showed how to check the rules without requiring the entire type hierarchy. They proposed the overloading rules in the context of a set of class declarations and function declarations without defining a language with complete static and dynamic semantics. Therefore, their work does not guarantee that the static overloading rules and the way a language chooses a method to invoke at run time match correctly. While FGFV does not support features like modularity, nominal exclusion, and closed types in their work, they can be added in an obvious way. As the related work section of [Allen et al. \[2011\]](#) discusses in detail, most earlier systems like *Dubious* [[Millstein and Chambers 1999, 2002](#); [Millstein 2003](#)], MultiJava [[Clifton et al. 2006](#)], and F(EML) [[Lee and Chambers 2006](#)] supported overloading with symmetric multiple dynamic dispatch with multiple inheritance, but without support for polymorphic methods or types. While $ML_{\leq}$ [[Bourdoncle and Merz 1997](#)] integrates polymorphism and symmetric multiple dynamic dispatch, it lacks multiple inheritance.

ParaSail [[Taft 2011, 2016](#)] is a parallel programming language designed to support inherently safe and secure, highly parallel applications. It also constrains the scope of type parameters, prohibiting a bound on a type parameter from referring to type parameter bindings to its left, so that dynamic type inference can be fast. However, ParaSail does not support covariant and contravariant type parameters, and it does not provide any formal specification.

Julia [[Bezanson et al. 2015, 2017](#)] is a dynamic programming language with symmetric multiple dispatch. It provides a type declaration and type annotation sublanguage, but does not have a static type system. Thus, it does not check the validity of overloaded methods, and only uses the subtype relation to find the most specific method among overloaded methods at run time.

Various languages provide different subsets of the FGFV features. Scala [Odersky et al. 2016] supports polymorphic types with variance, but it does not support symmetric multiple dispatch. FHJ [Wang et al. 2018] proposed a dispatch algorithm, which uses static types at run time to resolve overloading conflicts from unrelated methods that happen to have the same name and signature due to multiple inheritance. Castagna et al. [2015] presented a statically-typed calculus with dynamic (explicit) dispatch through pattern-matching, polymorphism, recursive types, semantic subtyping, and union, intersection, and negation type connectives. Their local type inference algorithm computes sets of type substitutions to insert in appropriate places in expressions. While their type-case expression and intersection type behave similarly to method overloading, they do not have a similar notion of overloading rules. Since the type-case expression executes the first matching expression, it does not introduce any ambiguity. For OCaml, Garrigue [2001; 2002] introduced a type inference system for structural polymorphism such as polymorphic records, polymorphic variants, and accurate matching. Type constraints are described by constraint domains that are defined separately from its type system, and the constraints contain some notion of lower and upper bounds, which respectively represent required and available labels or cases. He also extended his work with CoQ mechanization [Garrigue 2010, 2015]. The work of Castagna and Garrigue may serve as a possible starting point for finding the necessary substitution σ to satisfy the rule [E-SUB] for existential subtyping during static analysis. We believe that our work opens the gate to adding either variance annotation in the work of Castagna or a form of type-case in the work of Garrigue.

7 CONCLUSION

We defined FGFV, a core calculus for an object-oriented language with symmetric multiple dispatch, multiple inheritance, and parametric polymorphism with variance. In order to ensure type soundness of FGFV, it must guarantee no ambiguous method invocations at run time, and preserve type even when a method invocation is dispatched to a different method at run time from the method chosen at compile time. Thus, we formalized overloading rules using existential types and universal types quantified over method type parameters to mechanically check whether overloaded polymorphic methods are valid. As a result, in a well-formed program that satisfies the overloading rules, there always exists the single most specific method to invoke with respect to both static argument types and dynamic argument types. Further, with the static types of method invocation expressions delivered to run time, the dynamic semantics of FGFV that uses the new dynamic dispatch algorithm we presented is proven to make type-sound choices among applicable method declarations.

ACKNOWLEDGMENT

This work has received funding from National Research Foundation of Korea (NRF) (Grants NRF-2017R1A2B3012020 and 2017M3C4A7068177).

REFERENCES

- Eric Allen, David Chase, J. J. Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. 2008. The Fortress Language Specification Version 1.0.
- Eric Allen, J. J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. 2007. Modular Multiple Dispatch with Multiple Inheritance. In *Proceedings of the 2007 ACM Symposium on Applied Computing (SAC '07)*. ACM, New York, NY, USA, 1117–1121. <https://doi.org/10.1145/1244002.1244245>
- Eric Allen, Justin Hilburn, Scott Kilpatrick, Victor Luchangco, Sukyoung Ryu, David Chase, and Guy L. Steele Jr. 2011. Type-checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance. In *Proceedings of the ACM International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. ACM, New York, NY, USA, 973–992. <https://doi.org/10.1145/2048066.2048140>

- Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2015. Julia: A Fresh Approach to Numerical Computing (v4). *CoRR* abs/1411.1607 (July 2015), 1–37. <http://arxiv.org/abs/1411.1607>
- Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2017), 65–98. <https://doi.org/10.1137/141000671>
- François Bourdoncle and Stephan Merz. 1997. Type Checking Higher-Order Polymorphic Multi-methods. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 302–315. <https://doi.org/10.1145/263699.263743>
- Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1992. A Calculus for Overloaded Functions with Subtyping. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming (LFP '92)*. ACM, New York, NY, USA, 182–192. <https://doi.org/10.1145/141471.141537>
- Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. 1995. A Calculus for Overloaded Functions with Subtyping. *Information and Computation* 117, 1 (1995), 115–135. <https://doi.org/10.1006/inco.1995.1033>
- Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. 2015. Polymorphic Functions with Set-Theoretic Types: Part 2: Local Type Inference and Type Reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15)*. ACM, New York, NY, USA, 289–302. <https://doi.org/10.1145/2676726.2676991>
- Curtis Clifton, Gary T. Leavens, Craig Chambers, and Todd Millstein. 2000. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '00)*. ACM, New York, NY, USA, 130–145. <https://doi.org/10.1145/353171.353181>
- Curtis Clifton, Todd D. Millstein, Gary T. Leavens, and Craig Chambers. 2006. MultiJava: Design Rationale, Compiler Implementation, and Applications. *ACM Transactions on Programming Languages and Systems* 28, 3 (May 2006), 517–575. <https://doi.org/10.1145/1133651.1133655>
- Devin Coughlin and Bor-Yuh Evan Chang. 2014. Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '14)*. Association for Computing Machinery, New York, 73–85. <https://doi.org/10.1145/2535838.2535855>
- EPFL. 2017. A class for immutable linked lists representing ordered collections of elements of type 'A' [implementation of class List for the Scala programming language]. <https://github.com/scala/scala/blob/2.13.x/src/library/scala/collection/immutable/List.scala>. Accessed: 2017-11-27.
- Jacques Garrigue. 2001. Simple Type Inference for Structural Polymorphism. In *Proceedings of the Second Asian Workshop on Programming Languages and Systems (APLAS'01)*. 329–343. Also presented as [Garrigue 2002].
- Jacques Garrigue. 2002. Simple Type Inference for Structural Polymorphism. In *Ninth International Workshop on Foundations of Object-Oriented Languages*. 1–11. <http://www.cs.cmu.edu/~ldr/FOOL/papers9/garrigue.ps> Also presented as [Garrigue 2001].
- Jacques Garrigue. 2010. A Certified Implementation of ML with Structural Polymorphism. In *Programming Languages and Systems*, Kazunori Ueda (Ed.). Springer Berlin Heidelberg, 360–375. LNCS 6461. Proceedings of the Eighth Asian Symposium on Programming Languages and Systems (APLAS 2010), Shanghai, China. More complete journal version is [Garrigue 2015].
- Jacques Garrigue. 2015. A Certified Implementation of ML with Structural Polymorphism and Recursive Types. *Mathematical Structures in Computer Science* 25, 4 (May 2015), 867–891. <https://doi.org/10.1017/S0960129513000066> Special issue: selected papers from the Eighth Asian Symposium on Programming Languages and Systems (APLAS 2010), Shanghai, China. Extended version of [Garrigue 2010].
- Atsushi Igarashi and Mirko Viroli. 2006. Variant Parametric Types: A Flexible Subtyping Scheme for Generics. *ACM Transactions on Programming Languages and Systems* 28, 5 (September 2006), 795–847. <https://doi.org/10.1145/1152649.1152650>
- Jieung Kim and Sukyoung Ryu. 2011. Coq Mechanization of Featherweight Fortress with Multiple Dispatch and Multiple Inheritance. In *Certified Programs and Proofs (CPP '11)*, Jean-Pierre Jouannaud and Zhong Shao (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 264–279.
- Keunwoo Lee and Craig Chambers. 2006. Parameterized Modules for Classes and Extensible Functions. In *ECOOP 2006 – Object-Oriented Programming*, Dave Thomas (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 353–378.
- Todd Millstein and Craig Chambers. 1999. Modular Statically Typed Multimethods. In *ECOOP'99 – Object-Oriented Programming*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 279–303.
- Todd Millstein and Craig Chambers. 2002. Modular Statically Typed Multimethods. *Information and Computation* 175, 1 (2002), 76 – 118. <https://doi.org/10.1006/inco.2002.3103>
- Todd David Millstein. 2003. *Reconciling Software Extensibility with Modular Program Reasoning*. Ph.D. Dissertation. University of Washington.
- Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala* (3rd ed.). Artima Press, Walnut Creek, California, USA. ISBN 978-0981531687.

- Gyunghee Park, Jaemin Hong, Guy L. Steele Jr., and Sukyoung Ryu. 2018. Polymorphic Symmetric Multiple Dispatch with Variance (Extended Report).
- Guy Steele and David Chase. 2012. Personal communication.
- Guy L. Steele Jr. 2017. It's Time for a New Old Language. Published on YouTube. <https://youtu.be/7HKbjYqqPPQ> Video (TRT 1:12:11) of invited keynote talk at PPoPP '17: 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Austin, Texas, February 6, 2017.
- S. Tucker Taft. 2011. Multicore Programming in ParaSail: Parallel Specification and Implementation Language. In *Proceedings of the 16th Ada-Europe International Conference on Reliable Software Technologies (Ada-Europe'11)*. Springer-Verlag, Berlin, Heidelberg, 196–200. <http://dl.acm.org/citation.cfm?id=2018027.2018049>
- S. Tucker Taft. 2016. Designing ParaSail, a new programming language (blog). <http://parasail-programming-language.blogspot.com>. Accessed: 2018-04-09.
- Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. 2013. HALO: Haskell to Logic through Denotational Semantics. In *Proceedings of the 40th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery, New York, 431–442. <https://doi.org/10.1145/2429069.2429121>
- Yanlin Wang, Haoyuan Zhang, Bruno C. d. S. Oliveira, and Marco Servetto. 2018. FHJ: A Formal Model for Hierarchical Dispatching and Overriding. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*. 20:1–20:30.
- Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Information and Computation* 115, 1 (Nov. 1994), 38–94. <https://doi.org/10.1006/inco.1994.1093>