Type-checking Modular Multiple Dispatch with Parametric Polymorphism and Multiple Inheritance

Eric Allen
Oracle Labs
eric.allen@oracle.com

Justin Hilburn
Oracle Labs
justin.hilburn@oracle.com

Scott Kilpatrick
University of Texas at Austin
scottk@cs.utexas.edu

Victor Luchangco
Oracle Labs
victor.luchangco@oracle.com

Sukyoung Ryu
KAIST
sryu@cs.kaist.ac.kr

David Chase
Oracle Labs
david.r.chase@oracle.com

Guy L. Steele Jr.
Oracle Labs
guy.steele@oracle.com

Abstract

In previous work, we presented rules for defining overloaded functions to ensure type soundness and non-ambiguity of function calls under symmetric multiple dispatch in an object-oriented language with multiple inheritance, and we showed how to check these rules without requiring the entire type hierarchy to be known, thus supporting modularity and extensibility. In this work, we extend these rules to a language that supports parametric polymorphism on both classes and functions.

In a multiple-inheritance language in which any type may be extended by types in other modules, some overloaded functions that might seem valid are correctly rejected by our rules. We explain how these functions can be permitted in a language that supports an exclusion relation among types, allowing programmers to declare "nominal exclusions" and also implicitly imposing exclusion among different instances of each polymorphic type. We give rules for computing the exclusion relation, deriving many type exclusions from declared and implicit ones.

We show how to check the rules for overloaded functions, in particular, how extending the rules to handle parametric polymorphism reduces to the problem of determining subtyping relationships between universal and existential types. Our system has been implemented as part of the open-source Fortress compiler.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects, inheritance, modules, packages, polymorphism

General Terms Languages

Keywords object-oriented programming, multiple dispatch, symmetric dispatch, multiple inheritance, overloading, modularity, methods, multimethods, static types, run-time types, ilks, components, separate compilation, Fortress, meet rule

1. Introduction

A key feature of object-oriented languages is *dynamic dispatch*: there may be multiple definitions of a function (or method) with the same name—we say the function is *overloaded*—and a call to a function of that name is resolved based on the "run-time types"—we use the term *ilks*—of the arguments. With *single dispatch*, a particular argument is designated as the *receiver*, and the call is resolved only with respect to that argument. With *multiple dispatch*, the run-time types of all arguments to a call are used to resolve the call. *Symmetric multiple dispatch* is a special case of multiple dispatch in which all arguments are considered equally when resolving a call.

Multiple dispatch provides great expressivity. In particular, mathematical operators such as $+$ and $\leq$ and $\cup$ and especially $\cdot$ and $\times$ have different definitions depending on the types of the arguments to an application of the operator (even the number of arguments may vary between calls); in a language with multiple dispatch, it is natural to define these operators as overloaded functions. Similarly, many binary operations on collections such as *append* and *zip* have different definitions depending on the types of both arguments.
TODO: Add (reference to) argument for symmetric multiple dispatch?

In an object-oriented language with symmetric multiple dispatch, some restrictions must be placed on overloaded function definitions to guarantee type soundness and avoid ambiguous function calls. For example, consider the following overloaded function definitions:

$$f(b : B, a : A) : \mathbb{Z} = 1$$
$$f(a : A, b : B) : \mathbb{Z} = 2$$

If A is a subtype of B (we write this as $A <: B$), to which of these definitions do we dispatch when f is called with two arguments of type A?

Castagna *et al.* [4] address this problem in the context of a type system without parametric polymorphism or multiple inheritance by requiring every pair of overloaded function definitions to satisfy the following properties: (*i*) whenever the parameter type of one is a subtype of the parameter type of the other, the return type of the first must also be a subtype of the return type of the second; and (*ii*) whenever the parameter types of the two definitions have a common lower bound (i.e., nontrivial subtype), there is a unique definition for the same function whose parameter type is the greatest lower bound of the parameter types of the two definitions. Thus, for the example above, the programmer must provide a third definition to satisfy the latter property:

$f(a : A, a' : A) : \mathbb{Z} = \ldots$

This latter property is equivalent to requiring that the definitions for each overloaded function form a meet semilattice partially ordered by the subtype relation on their parameter types. We call this partial order the *more specific than* relation,[1] and we call the property the *meet rule*. We call the first property above the *return type rule* or *subtype rule*.

In this paper, we give variants of these rules for ensuring safe overloaded functions in a language that supports symmetric multiple dispatch, multiple inheritance, and parametric polymorphism (that is, generic types *and* generic functions). We prove that these rules guarantee type soundness and that there are no ambiguous calls at run time (see Section 4). We do this by extending our earlier rules for a core of the Fortress programming language that did not support generics [1, 2], for which we proved the analogous theorems.

The type system considered by Castagna *et al.* assumed knowledge of the entire type hierarchy (to determine whether two types have a common subtype), and the type hierarchy was assumed to be a meet semilattice (to ensure that any two types have a greatest lower bound). In previous work [2], we applied variants of the meet and return type rules to a simplified version of the Fortress programming language [1], which supports multiple inheritance and does not require that types have expressible meets (i.e., the types that can be expressed in the language need not form a meet semilattice). We showed that we could check these rules in a modular way, so that the type hierarchy could be extended safely by new modules without rechecking old modules.

Because the type hierarchy defined by a module may be extended, and because Fortress supports multiple inheritance, two types may have a common nontrivial subtype even if no declared type extends them both. Thus, for any pair of overloaded function definitions with incomparable parameter types (i.e., neither definition is more specific than the other), the meet rule requires some other definition to resolve the potential ambiguity. Because explicit intersection types cannot be expressed in Fortress, it is not always possible to provide such a function definition. However, Fortress defines an *exclusion relation* on types, such that types related by exclusion must have no common nontrivial subtypes, and thus definitions with such types as parameter types need not be disambiguated.

In this paper, we extend our prior rules to handle parametric polymorphism, where both types and functions may be parameterized by type variables, which Fortress also supports. To do so, it is helpful to have an interpretation for parametric types and type-parametric functions.

One way to think about a parametric type such as $\mathrm{List}[\![T]\!]$ (a list with elements of type T—type parameter lists in Fortress are delimited by white square brackets) is that it represents an infinite set of ground types $\mathrm{List}[\![\mathrm{Object}]\!]$ (lists of objects), $\mathrm{List}[\![\mathrm{String}]\!]$ (lists of strings), $\mathrm{List}[\![\mathbb{Z}]\!]$ (lists of integers), and so on. An actual type checker must have rules for working with uninstantiated (non-ground) parametric types, but for many purposes this model of "an infinite set of ground types" is adequate for explanatory purposes. Not so, however, for type-parametric functions.

For some time during the development of Fortress, one of us (Steele) pushed for an interpretation of type-parametric functions analogous to the one above for parametric types; that is, that the type-parametric function definition[2]

$$append[\![T]\!](x : \mathrm{List}[\![T]\!], y : \mathrm{List}[\![T]\!]) : \mathrm{List}[\![T]\!] = e$$

should be understood as if it stood for an infinite set of monomorphic definitions:

$$append(x : \mathrm{List}[\![\mathrm{Object}]\!], y : \mathrm{List}[\![\mathrm{Object}]\!]) : \mathrm{List}[\![\mathrm{Object}]\!] = e$$
$$append(x : \mathrm{List}[\![\mathrm{String}]\!], y : \mathrm{List}[\![\mathrm{String}]\!]) : \mathrm{List}[\![\mathrm{String}]\!] = e$$
$$append(x : \mathrm{List}[\![\mathbb{Z}]\!], y : \mathrm{List}[\![\mathbb{Z}]\!]) : \mathrm{List}[\![\mathbb{Z}]\!] = e$$
$$\ldots$$

The intuition was that for any specific function call, the usual rule for dispatch would then choose the appropriate most specific definition for this (infinitely) overloaded function.

Although that intuition worked well enough for a single polymorphic function definition, it failed utterly when we considered multiple function definitions. For example, a programmer might want to provide definitions for specific monomorphic special cases, as in:

$$append[\![T]\!](x : \mathrm{List}[\![T]\!], y : \mathrm{List}[\![T]\!]) : \mathrm{List}[\![T]\!] = e_1$$
$$append(x : \mathrm{List}[\![\mathbb{Z}]\!], y : \mathrm{List}[\![\mathbb{Z}]\!]) : \mathrm{List}[\![\mathbb{Z}]\!] = e_2$$

But if the interpretation above is taken seriously, this would be equivalent to:

$$append(x : \mathrm{List}[\![\mathrm{Object}]\!], y : \mathrm{List}[\![\mathrm{Object}]\!]) : \mathrm{List}[\![\mathrm{Object}]\!] = e_1$$
$$append(x : \mathrm{List}[\![\mathrm{String}]\!], y : \mathrm{List}[\![\mathrm{String}]\!]) : \mathrm{List}[\![\mathrm{String}]\!] = e_1$$
$$append(x : \mathrm{List}[\![\mathbb{Z}]\!], y : \mathrm{List}[\![\mathbb{Z}]\!]) : \mathrm{List}[\![\mathbb{Z}]\!] = e_1$$
$$\ldots$$
$$append(x : \mathrm{List}[\![\mathbb{Z}]\!], y : \mathrm{List}[\![\mathbb{Z}]\!]) : \mathrm{List}[\![\mathbb{Z}]\!] = e_2$$

and we can see that there is an ambiguity when the arguments are both of type $\mathrm{List}[\![\mathbb{Z}]\!]$.

[1] Despite its name, this relation, like the subtype relation, is reflexive: two function definitions with the same parameter type are each more specific than the other. In that case, we say the definitions are equally specific.

[2] The first pair of white square brackets delimits the declaration of a type parameter T, but the other pairs of white brackets indicate that this type variable T is the static argument to the parametric type List.

It gets worse if the programmer wishes to handle an infinite set of cases specially. It would seem natural to write

$$append[\![T]\!](x\colon \mathrm{List}[\![T]\!], y\colon \mathrm{List}[\![T]\!])\colon \mathrm{List}[\![T]\!] = e_1$$
$$append[\![T <\colon \mathrm{Number}]\!](x\colon \mathrm{List}[\![T]\!], y\colon \mathrm{List}[\![T]\!])\colon \mathrm{List}[\![T]\!] = e_2$$

to handle specially all cases where T is a subtype of Number. But the model would regard this as an overloading with an infinite number of ambiguities.

To resolve this problem, we had to develop an alternate model and an associated type system that could handle overloaded type-parametric functions in a manner that would accord with programmer intuition and support the plausible examples shown above.

Two other authors of this paper

The key insight

Credit for championing key insights—regarding each polymorphic definition as a single definition (rather than an infinite set of definitions) competing in the overload set, and using universal and existential types to describe them in the type system (an idea reported by Bourdoncle and Merz [3])—belongs to two other authors of this paper (Hilburn and Kilpatrick). Adopting this new approach has made overloaded polymorphic functions both tractable and effective for writing Fortress code.

In this paper, we give rules for ensuring safe overloaded functions in a language that supports symmetric multiple dispatch, multiple inheritance, and parametric polymorphism (that is, generic types *and* generic functions), and we prove that these rules guarantee that there are no ambiguous calls at run time (see Section 4). We do this by extending our earlier rules for a core of the Fortress programming language that did not support generics [1, 2], for which we proved the analogous theorem. To minimize syntactic overhead and avoid having to translate between a concrete language syntax and a formal semantics, we present these rules (see Section 3) in the context of a straightforward formalization of a type system supporting multiple inheritance and parametric polymorphism, which we define in Section 2.

The problem of dynamic dispatch in the presence of overloaded *generic* functions is challenging because the overloaded definitions might have not only distinct argument types, but also distinct type parameters (even different numbers of type parameters), so the type values of these parameters make sense only in distinct type environments. For example, consider the following overloaded function definitions in Fortress:

$$combine[\![T]\!](xs\colon \mathrm{List}[\![T]\!], ys\colon \mathrm{List}[\![T]\!])\colon \mathrm{List}[\![T]\!]$$
$$combine[\![S, T]\!](s\colon \mathrm{Table}[\![S, T]\!], t\colon \mathrm{Table}[\![S, T]\!])\colon \mathrm{Table}[\![S, T]\!]$$

The first definition declares a single type parameter denoting the types of the elements of the two list arguments xs and ys. The second definition declares two type parameters corresponding to the domains and ranges of the two table arguments s and t. But the type parameter of the first definition bears no relation to the type parameters of the second. How should we compare such function definitions to determine which is the best to dispatch to? How can we ensure that there even is a best one in all cases? Furthermore, the rules must be compatible with type inference, since instantiation of type parameters at a call site is typically done automatically. So even determining which definitions are applicable to a particular call is not always obvious.

In providing rules to ensure that any valid set of overloaded function definitions guarantees that there is always a unique function to call at run time, we strive to be maximally permissive: A set of overloaded definitions should be disallowed only if it permits ambiguity that cannot be resolved at run time. Nonetheless, we show in Section 5 that some seemingly valid sets of overloaded functions are rejected by our rules, and rightly so: although intuitively appealing, these overloaded functions admit ambiguous calls.

Many of these overloaded functions can, and we believe should, be allowed if the type system supports an *exclusion relation*, which asserts that two types have no common instances. If the domains of two function definitions exclude each other, then these definitions can never be applicable to the same call, and so no ambiguity can arise between them. Many languages provide a way of declaring some exclusion relations implicitly. For example, single inheritance ensures that, for any two types, if one is not a subtype of the other, then the two types exclude each other. Fortress enables programmers to declare "nominal exclusion" in addition to determining many exclusions implicitly, and in Section 6, we formalize how Fortress does this, and show how this exclusion relation is used to improve expressivity by accommodating overloadings that would otherwise be rejected. The proof of safety in Section 4 covers the rules under this extended type system.

In Section 8, we discuss type inference and modularity. We discuss related work in Section 9 and conclude in Section 10.

2. Preliminaries

2.1 Types

Following Kennedy and Pierce [8], we define a world of types ranged over by metavariables S, T, U, V, and W. Types are of four forms: *type variables* (ranged over by metavariables X, Y, and Z); *constructed types* (ranged over by metavariables K, L, M and N), written $C[\![\overline{T}]\!]$ where C is a type constructor and $\overline{T}$ is a list of types; *structural types*, consisting of arrow and tuple types; and *compound types*, consisting of intersection and union types. In addition, there are two special constructed types, Any and $\mathrm{BottomType}$, explained below. ***TODO***: Make $\mathrm{BottomType}$ special, not a constructed type. The abstract syntax of types is defined in BNF as follows (where $\overline{A}$ indicates a possibly empty comma-separated se-

quence of syntactic elements A):

$T ::= X$		type variable
	$C[\![\overline{T}]\!]$	type constructor application
	$T \to T$	arrow type
	$(\overline{T})$	tuple type
	$T \cap T$	intersection type
	$T \cup T$	union type
	Any	
	$\mathrm{BottomType}$	

A tuple type of length one is synonymous with its element type. A tuple type with any $\mathrm{BottomType}$ element is synonymous with $\mathrm{BottomType}$. As in Fortress, compound types—intersection and union types—and $\mathrm{BottomType}$ are *not* first-class: they cannot be written in a program; rather, they are used by the type analyzer during type checking. For example, type variables may have multiple bounds, so that any valid instantiation of such a variable must be a subtype of the intersection of its bounds.

TODO: Only considering ground types at first. (Define ground types.)

Constructed types (other than Any and $\mathrm{BottomType}$) are applications of *type constructors*. A *class table* $\mathcal{T}$ is a set of type constructor declarations (at most one declaration for each type constructor) of the following form:

$$C[\![\overline{X <: \{\overline{M}\}}]\!] <: \{\overline{N}\}$$

This declaration indicates that an application $C[\![\overline{U}]\!]$ (*i*) is *well-formed* (with respect to $\mathcal{T}$) if and only if $|\overline{U}| = |\overline{X}|$ and for all i and j, $U_i <: [\overline{U}/\overline{X}]M_{ij}$ for each bound M_{ij} (where $<:$ is the subtyping relation defined below, and $[\overline{U}/\overline{X}]M_{ij}$ is M_{ij} with U_k substituted for each occurrence of X_k in M_{ij} for $1 \le k \le |\overline{U}|$); and (*ii*) is a subtype of $[\overline{U}/\overline{X}]N_l$ for $1 \le l \le |\overline{N}|$. Thus, a class table induces a (nominal) *subtyping relation* over the constructed types by taking the reflexive and transitive closure of the subtyping relation derived from the declarations in the class table. In addition, every type is a subtype of Any and a supertype of $\mathrm{BottomType}$. For any type T (of any form), we write $T \in \mathcal{T}$ to mean that any constructed type occurring in type T is well-formed with respect to $\mathcal{T}$.

A class table $\mathcal{T}$ is *well-formed* if the resulting subtyping relation on its constructed types is a partial order. As usual for languages with nominal subtyping, we allow recursive and mutually recursive references in $\mathcal{T}$.

TODO: What does "recursive and mutually recursive references" mean?

A class table $\mathcal{T}'$ is an *extension* of $\mathcal{T}$ (written $\mathcal{T}' \supseteq \mathcal{T}$) if every constructor declaration in $\mathcal{T}$ is also in $\mathcal{T}'$ and the subtype relation on $\mathcal{T}'$ agrees with that of $\mathcal{T}$. Consequently, if $\mathcal{T}' \supseteq \mathcal{T}$ then, for all types T, $T \in \mathcal{T}$ implies $T \in \mathcal{T}'$. We typically omit explicit reference to the class table when it is understood, and we assume that the class table is well-formed.

Given a well-formed class table containing the type constructor declaration for C above and a well-formed application $C[\![\overline{T}]\!]$, we denote the set of its explicitly declared supertypes by

$$C[\![\overline{T}]\!].\mathit{extends} = \{\overline{[\overline{T}/\overline{X}]N}\}$$

and the set of ancestors of $C[\![\overline{T}]\!]$ (defined recursively) by

$$\mathit{ancestors}(C[\![\overline{T}]\!]) = \{C[\![\overline{T}]\!]\} \cup \bigcup_{M \in C[\![\overline{T}]\!].\mathit{extends}} \mathit{ancestors}(M).$$

(To reduce clutter, nullary applications are written without brackets; for example, $C[\![\,]\!]$ is written C.) ***TODO***: Other for Any and Botto

Structural and compound types are *well-formed* (with respect to a class table) if their constituent types are well-formed. We extend the subtyping relation to structural and compound types in the usual way: Arrow types are contravariant in their domain types and covariant in their range types (i.e., $S \to T <: U \to V$ if and only if $U <: S$ and $T <: V$). One tuple type is a subtype of another if and only if they have the same number of elements, and each element of the first is a subtype of the corresponding element of the other (i.e., $(\overline{S}) <: (\overline{T})$ if and only if $|\overline{S}| = |\overline{T}|$ and $S_i <: T_i$ for all $1 \le i \le |\overline{S}|$). An intersection type is by definition the most general type that is a subtype of each of its element types: $(A \cap B) <: A$, $(A \cap B) <: B$, and for all types T, if $T <: A$ and $T <: B$ then $T <: (A \cap B)$. Similarly, a union type is by definition the most specific type that is a supertype of each of its element types: $A <: (A \cup B)$, $B <: (A \cup B)$, and for all types T, if $A <: T$ and $B <: T$ then $(A \cup B) <: T$.

To extend the subtyping relation to type variables, we require a *type environment*, which maps type variables to bounds:

$$\Delta = \overline{X <: \{\overline{M}\}}$$

In the context of Δ, each type variable X_i is a subtype of each of its bounds M_{ij}. Note that the type variables X_i may appear within the bounds M_{ij}. We write $\Delta \vdash S <: T$ to indicate the judgment that S is a subtype of T in the context of Δ. When Δ is understood to be empty, we write this judgment as simply $S <: T$. The subtype judgment can only be made on types $S, T \in \mathcal{T}$, so the judgment takes a class table $\mathcal{T}$ as an implicit parameter. The types S and T are said to be *equivalent*, written $S \equiv T$, when $S <: T$ and $T <: S$.

To allow separate compilation of program components, we do not assume that the class table is complete; there might be declarations yet unknown. Specifically, we cannot infer that two constructed types have no common constructed subtype (other than $\mathrm{BottomType}$) from the lack of any such type in the class table. However, we do assume that each declaration is complete, and furthermore, that any type constructor used in the class table (e.g., in a bound or a supertype of another declaration) is declared in the table, so that all the supertypes of a constructed type are known.

2.2 Values and Ilks

Types are intended to describe the values that might be produced by an expression or passed into a function. In Fortress,

for example, there are three kinds of values: objects, functions, and tuples; every object belongs to at least one constructed type, every function belongs to at least one arrow type, and every tuple belongs to at least one tuple type. When we say that two types T and U have *the same extent*, we mean that for every value v, v belongs to T if and only if v belongs to U.

We place a requirement on values and on the type system that describes them: Although a value may belong to more than one type, for every value v there is a unique type $ilk(v)$ (the *ilk* of the value) that is *representable in the type system*[3] and has the property that for every type T, if v belongs to T then $ilk(v) <: T$; moreover, $ilk(v) \neq \mathrm{BottomType}$. (This notion of ilk corresponds to what is sometimes called the "class" or "run-time type" of the value.[4])

The implementation significance of ilks is that it is possible to select the dynamically most specific applicable function from an overload set using only the ilks of the argument values; no other information about the arguments is needed.

In a sound type system, if an expression is determined by the type system to have type T, then every value computed by the expression at run time will belong to type T; moreover, whenever a function whose ilk is $U \to V$ is applied to an argument value, then the argument value will belong to type U.

2.3 Overloaded Functions

A function declaration consists of a name, a sequence of type parameter declarations (enclosed in white square brackets), a type indicating the domain of the function, and a type indicating the range of the function. A type parameter declaration consists of a type parameter name and its bounds. We omit the white square brackets of a declaration when the sequence of type parameter declarations is empty. The abstract syntax of function declarations is as follows:

$$\begin{array}{lcl} \mathit{Decl} & ::= & \mathit{Id}[\![\overline{\mathit{Id} <: \{\overline{\mathit{Type}}\}}]\!]\ \mathit{Type} : \mathit{Type} \\ & | & \mathit{Id}\ \mathit{Type} : \mathit{Type} \end{array}$$

For example, in the following function declaration:

$$f[\![X <: M, Y <: N]\!]\big(\mathrm{List}[\![X]\!], \mathrm{Tree}[\![Y]\!]\big) : \mathrm{Map}[\![X, Y]\!]$$

the name of the function is f, the type parameter declarations are $X <: M$ and $Y <: N$, the domain type is $\big(\mathrm{List}[\![X]\!], \mathrm{Tree}[\![Y]\!]\big)$, which is a tuple type, and the range type is $\mathrm{Map}[\![X, Y]\!]$. We will often abbreviate a function as $f[\![\Delta]\!]\ S : T$ when we do not want to emphasize the bounds (we are abusing notation by letting Δ range over both type environments and bounds definitions).

A function declaration $f[\![\overline{X <: \{\overline{N}\}}]\!]\ S : T$ may be *instantiated* with type arguments $\overline{W}$ if $|\overline{W}| = |\overline{X}|$ and $W_i <: [\overline{W}/\overline{X}]N_{ij}$ for all i and j; we call $[\overline{W}/\overline{X}]f\ S : T$ the *instantiation* of f with $\overline{W}$. When we do not care about $\overline{W}$, we just say that $f\ U : V$ is an *instance* of f (and it is understood that $U = [\overline{W}/\overline{X}]S$ and $V = [\overline{W}/\overline{X}]T$ for some (unstated) $\overline{W}$). We use the metavariable $\mathcal{D}$ for a finite collection of sets of function declarations and $\mathcal{D}(f)$ for the set in $\mathcal{D}$ that contains all declarations named f. An instance $f\ U : V$ of a declaration f is *applicable* to a type T if and only if $T <: U$. A function declaration is *applicable* to a type if and only if at least one of its instances is. For any two function declarations $f_1, f_2 \in \mathcal{D}(f)$, f_1 is *more specific* than f_2 (written $f_1 \preceq f_2$) if and only if for every type T such that f_1 is applicable to T, f_2 is also applicable to T.

3. Parametrically Polymorphic Overloading

Guaranteeing valid overloading requires constraints on the sets of overloaded function declarations that are allowed to appear in a legal program [11, 12]. Some languages require program constructs that encapsulate all overloaded definitions; such constructs essentially guarantee valid overloading "by construction". Examples in other languages include type classes [6, 16, 19] and multimethods [3, 11, 12]. We take a different approach: Rather than introducing additional language facilities, we impose rules on the function declarations themselves. We intend these rules to be "minimal" in that they should be as unrestrictive as possible while preserving the ability to guarantee valid overloading and be checked in a modular way. We took a similar approach to guarantee safety for overloaded monomorphic functions [2]. However, handling parametric polymorphism and implicit instantiation (i.e., type inference) requires more sophisticated type analysis. We achieve this analysis by introducing universal and existential quantification over the ground types defined by class tables.

Specifically, in this section, we define three rules—the *No Duplicates Rule*, *Meet Rule*, and *Return Type Rule*—for sets of overloaded function definitions, and say that such a set is *well-formed with respect to a class table* if it satisfies all these rules using the subtyping relation induced by the class table. We describe how to mechanically verify these rules in a modular way in terms of subtyping relations on universal and existential types in Section 7, and we show that any valid set of overloaded function declarations is safe in Section 4.

3.1 Overloading Rules

In this section, we describe the rules for valid overloading. For each function name f, we determine whether a set of overloaded function declarations $\mathcal{D}(f)$ is valid by independently considering every pair of declarations in the set. A

[3] The type system presented here satisfies this requirement simply by providing intersection types. The Fortress type system happens to satisfy it in another way as well, which is typical of object-oriented language designs: every object is created as an instance of a single nominal constructed type, and this type is its ilk.

[4] We prefer the term "ilk" to "run-time type" because the notion—and usefulness—of the most specific type to which a value belongs is not confined to run time. We prefer it to the term "class," which is used in *The Java Language Specification* [7], because not every language uses the term "class" or requires that every value belong to a class. For those who like acronyms, we offer the mnemonic retronyms "implementation-level kind" and "intrinsically least kind."

pair of declarations is a valid overloading if it satisfies one of three rules described below.

To avoid the obvious ambiguity, $\mathcal{D}(f)$ should not contain equally specific declarations: for each pair of overloaded declarations, either one declaration is strictly more specific than the other or they are incomparable.

No Duplicates Rule $\mathcal{D}(f)$ does not contain any two declarations that are equally specific. In other words, there are no (distinct) declarations $f_1, f_2 \in \mathcal{D}(f)$ such that $f_1 \preceq f_2$ and $f_2 \preceq f_1$.

A pair of declarations is a valid overloading if for any call to which both declarations are applicable, there is a *disambiguating declaration* (possibly one of the pair) that is also applicable to the call and is at least as specific as both declarations. Thus, at run time, the disambiguating declaration is preferred.

Meet Rule For each pair of declarations $f_1, f_2 \in \mathcal{D}(f)$, and for every type $T \in \mathcal{T}$, there should exist a third declaration $f_0 \in \mathcal{D}(f)$ (possibly one of the pair) such that f_0 is applicable to T if and only if both f_1 and f_2 are applicable to T.

If one monomorphic declaration is more specific than another monomorphic one then there is no ambiguity between these two declarations: for any call to which both are applicable, the first is more specific. In the parametrically polymorphic setting, if one declaration is to be regarded as more specific than another, we require that for every instance of the second that is applicable to a call, there exist an instance of the first that is also applicable to the call. As in other object-oriented languages, to ensure type safety in the face of dynamic dispatch, we also require that the return type of the latter declaration be a subtype of the return type of the former.

Return Type Rule For every $f_1, f_2 \in \mathcal{D}(f)$ with $f_1 \preceq f_2$, for every type W to which f_1 is applicable (and therefore f_2 is also applicable) and every instance $f_2'\, S_2' : T_2'$ of f_2 that is applicable to W, there must exist an instance $f_1'\, S_1' : T_1'$ of f_1 that is applicable to W and satisfies $T_1' <: T_2'$.

4. Overloading Resolution Safety

Lemma 1 (Progress) If some declaration in $\mathcal{D}(f)$ is applicable to W then there is a unique most specific declaration $f_W \in \mathcal{D}(f)$ that is applicable to W.

Proof Sketch: Our proof strategy for satisfying the Progress condition makes use of the old idea from Castagna *et al.* [4] that for each name f the set of function declarations $\mathcal{D}(f)$ should form a meet semilattice under the *specificity* order defined in Section 2.3. If any declaration of name f is applicable to a type W, then the set $\mathcal{D}_W(f) \subseteq \mathcal{D}(f)$ of all declarations named f and applicable to W also forms a nonempty meet semilattice under specificity. Therefore $\mathcal{D}_W(f)$ must have a least element. □

Lemma 2 (Preservation) If $f\, S : T$ is an instance of some declaration and $f\, S : T$ is applicable to W, then there exists some instance $f_W\, U : V$ of f_W such that $f_W\, U : V$ is applicable to W and $V <: T$.

Proof Sketch: Note that subtyping is preserved under class table extension, so if $\mathcal{D}(f)$ satisfies the No Duplicates Rule and the Meet Rule with respect to the class table $\mathcal{T}$ then it satisfies them with respect to $\mathcal{T}'$ for any $\mathcal{T}' \supseteq \mathcal{T}$. Therefore, we can be certain that adding more types will not invalidate the Progress guarantee. Just as in our discussion of the Meet Rule, the fact that subtyping is preserved under class table extension makes sure that the property that $\mathcal{D}(f)$ satisfies the Return Type Rule is preserved under class table extension. Therefore, the Return Type Rule and the rules from the last section are sufficient to ensure safety. □

Theorem 1 (Overloading Resolution Safety) There exists always a unique function to call at run time among a collection of overloaded function declarations.

Proof Sketch: Lemma 1 ensures that we never get stuck resolving an application; every call to a function declaration is unambiguous. Lemma 2 ensures that the static type of an application is a supertype of the ilk of each value produced by the application at run time, provided that the ilks for the argument values in function applications are always subtypes of the static types of the argument expressions (as they are in any language with a sound type system). □
Note that as a monomorphic function declaration is a special case of a generic function declaration, where the sequence of the type parameters is empty, these conditions apply also for monomorphic function declarations.

5. Disallowed Valid Overloading

While the rules presented in Section 3 allow programmers to write valid sets of overloaded generic function declarations, they sometimes reject "seemingly" valid overloadings. In fact, these are not false negatives; many declarations that programmers would like to write are actually unsafe due to multiple inheritance.

For example, even though the following function declarations look like a valid overloading, they are not because the Meet Rule is not satisfied:

$$simple\ \mathrm{String} : \mathrm{String}$$
$$simple\ \mathbb{Z} : \mathbb{Z}$$

Moreover, in Fortress it is impossible to disambiguate the declarations by providing the meet because intersection types are not allowed in the Fortress syntax. In a language with single inheritance, we might infer that these overloadings were safe because a class can only have a single superclass. However, due to multiple inheritance, we cannot

be sure that these types do not have a common subtype, no matter what the programmer intends.

Now consider this less trivial set of overloaded functions:

$$foo[\![X <: \mathrm{Any}]\!]\mathrm{ArrayList}[\![X]\!]: \mathbb{Z}$$
$$foo[\![Y <: \mathbb{Z}]\!]\mathrm{List}[\![Y]\!]: \mathbb{Z}$$
$$foo[\![W <: \mathbb{Z}]\!]\mathrm{ArrayList}[\![W]\!]: \mathbb{Z}$$

where $\mathrm{ArrayList}[\![T]\!] <: \mathrm{List}[\![T]\!]$ for all types T.[5] The first two declarations are incomparable under specificity—the first declaration applies to all instantiations of type constructor $\mathrm{ArrayList}$, whereas the second declaration applies only to instantiations of type constructor List with subtypes of type $\mathbb{Z}$. The third definition, which is the "obvious" candidate to disambiguate the two, is not actually the meet; the domain of this meet candidate is the existential type:

$$\exists[\![W <: \mathbb{Z}]\!]\mathrm{ArrayList}[\![W]\!]$$

and needs to be proven equivalent to the domain of the computed meet:

$$\exists[\![X <: \mathrm{Any}, Y <: \mathbb{Z}]\!]\big(\mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!]\big)$$

which requires that the latter be a subtype of the former. However, there is no type $W <: \mathbb{Z}$ such that:

$$X <: \mathrm{Any}, Y <: \mathbb{Z} \vdash \mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!] <: \mathrm{ArrayList}[\![W]\!]$$

Our definition of the meet is not faulty: these declarations actually are unsafe. Consider the (user-defined) constructed type:

$$\mathrm{BadList} <: \big\{\mathrm{ArrayList}[\![\mathrm{String}]\!], \mathrm{List}[\![\mathbb{Z}]\!]\big\}$$

Our meet candidate is not applicable to $\mathrm{BadList}$, but the two other definitions of *foo* are. Since neither of those is more specific than the other, this set of overloaded declarations must be rejected.

The following overloading example, in which the second declaration is more specific than the first, is also ill-formed:

$$tail[\![X <: \mathrm{Any}]\!]\mathrm{List}[\![X]\!]: \mathrm{List}[\![X]\!]$$
$$tail[\![Y <: \mathrm{Any}]\!]\mathrm{ArrayList}[\![Y]\!]: \mathrm{ArrayList}[\![Y]\!]$$

The declarations do not satisfy the Return Type Rule because we cannot find a specific type $V <: \mathrm{Any}$ such that:

$$X <: \mathrm{Any}, Y <: \mathrm{Any} \vdash \mathrm{ArrayList}[\![V]\!] \to \mathrm{ArrayList}[\![V]\!] <: \mathrm{ArrayList}[\![Y]\!] \cap \mathrm{List}[\![X]\!] \to \mathrm{List}[\![X]\!]$$

Once again, the type $\mathrm{BadList}$ proves that this set of overloaded declarations must be rejected. Consider the instance:

$$tail\,\mathrm{List}[\![\mathbb{Z}]\!]: \mathrm{List}[\![\mathbb{Z}]\!]$$

of the first declaration. We need to find an instance of the second declaration that is applicable to $\mathrm{BadList}$ and has a return type that is a subtype of $\mathrm{List}[\![\mathbb{Z}]\!]$, but the only instance of the second declaration applicable to $\mathrm{BadList}$ is:

$$tail\,\mathrm{ArrayList}[\![\mathrm{String}]\!]: \mathrm{ArrayList}[\![\mathrm{String}]\!]$$

whose return type is not a subtype of $\mathrm{List}[\![\mathbb{Z}]\!]$. Therefore, this set of overloaded declarations must be rejected.

[5] We use this standard declaration for $\mathrm{ArrayList}$ throughout.

6. More Permissive Overloading

To allow more overloaded functions such as *simple*, *tail*, and *foo* described in Section 5, we extend the overloading rules using *exclusion* relations on types.

6.1 Type Exclusion

To provide more expressive power to describe richer type relationships, we augment our formalism with an *exclusion* relation $\lozenge$ on types: $S \lozenge T$ asserts that types S and T have no common subtypes other than $\mathrm{BottomType}$. This allows us to describe explicitly what is typically only implicit in single-inheritance class hierarchies.

We define the exclusion relation by extending type constructor declarations with two new optional clauses, `excludes` and `comprises`:

$$C[\![\overline{X <: \{\overline{L}\}}]\!] <: \{\overline{N}\}\ \big[\texttt{excludes}\ \{\overline{M}\}\big]\ \big[\texttt{comprises}\ \{\overline{K}\}\big]$$

We also allow another form of declaration that is frequently convenient for defining "leaf types":

$$\texttt{object}\ C[\![\overline{X <: \{\overline{L}\}}]\!] <: \{\overline{N}\}$$

The exclusion relation on constructed types can then be described in terms of more precise sub-relations on those types, each of which corresponds to a certain reason for (or proof of) exclusion:

1. The `excludes` clause explicitly states that the constructed type $C[\![\overline{T}]\!]$ excludes $[\overline{T}/\overline{X}]M_i$ for each M_i in $\overline{M}$, which implies that any subtype of $C[\![\overline{T}]\!]$ also excludes each $[\overline{T}/\overline{X}]M_i$. We write this exclusion sub-relation as $C[\![\overline{T}]\!] \triangleright_{\mathrm{e}} [\overline{T}/\overline{X}]M_i$.
2. The `comprises` clause stipulates that any subtype of $C[\![\overline{T}]\!]$ *must* be a subtype of $[\overline{T}/\overline{X}]K_i$ for some K_i in $\overline{K}$. Then if every $[\overline{T}/\overline{X}]K_i$ in $\overline{K}$ excludes some type U, $C[\![\overline{T}]\!]$ must also exclude U. We write this exclusion sub-relation as $C[\![\overline{T}]\!] \triangleright_{\mathrm{c}} U$.
3. The `object` keyword denotes a type constructor whose applications have no non-trivial subtypes; an `object` type constructor is a leaf of the class hierarchy. Since such a constructed type $C[\![\overline{T}]\!]$ has no subtypes other than itself and $\mathrm{BottomType}$, we know that it excludes any type U of which it is not a subtype. We write this exclusion sub-relation as $C[\![\overline{T}]\!] \triangleright_{\mathrm{o}} U$.

We take the symmetric closure of each of these relations to get the relations $\lozenge_{\mathrm{e}}$, $\lozenge_{\mathrm{c}}$ and $\lozenge_{\mathrm{o}}$. Exclusion between constructed types is informally defined as the union of these symmetric relations. (We introduce another sub-relation $\lozenge_{\mathrm{p}}$ in Section 6.3.)

We can extend the exclusion relation to structural and compound types as follows: Every arrow type excludes every non-arrow type. Every singleton tuple type excludes exactly those types excluded by its element type. Every non-singleton tuple type excludes every non-tuple type. Tuple type $(\overline{V})$ excludes $(\overline{W})$ if either $|\overline{V}| \neq |\overline{W}|$ or V_i excludes W_i for some i. An intersection type excludes any type excluded by *any* of its constituent types, while a union type excludes any type excluded by *all* of its constituent types. $\mathrm{BottomType}$ excludes every type (including itself—it is the only type that excludes itself), and Any does not exclude any type other than $\mathrm{BottomType}$. (We define the exclusion relation formally in Figure 1 in Section 6.4.)

We augment our notion of a well-formed class table to require that the subtyping and exclusion relations it induces "respect" each other. That is, for all constructed types M and N other than $\mathrm{BottomType}$,

1. If M excludes N then M must not be a subtype of N.
2. If $N <: M$ and M has a `comprises` clause, then there is some constructed type K in the `comprises` clause of M such that $N <: K$.
3. If the type constructor C is declared as an `object`, then its `comprises` clause must be empty, and there is no other constructed type N such that $N <: C[\![\overline{T}]\!]$ for any types $\overline{T}$.

As with the subtyping relation, a valid extension to a class table $\mathcal{T}$ must preserve these well-formedness properties.

For convenience, we allow the `excludes` and `comprises` clauses to be omitted. Omitting an `excludes` clause is equivalent to having `excludes` $\{\}$, and omitting a `comprises` clause is equivalent to having `comprises` $\{\mathrm{Any}\}$.[6] For an application $C[\![\overline{T}]\!]$ of a declaration with the `excludes` and/or `comprises` clause above, we define the sets of instantiations of the types in these clauses analogously to $C[\![\overline{T}]\!].\mathit{extends}$. That is,

$$C[\![\overline{T}]\!].\mathit{excludes} = \{\overline{[\overline{T}/\overline{X}]M}\}$$
$$C[\![\overline{T}]\!].\mathit{comprises} = \{\overline{[\overline{T}/\overline{X}]K}\}$$

6.2 Overloading Rule with Exclusion

If the parameter types of a set of overloaded functions are disjoint, the set does not introduce ambiguity: they are never applicable to the same call. Therefore, to allow the function simple as a valid overloading, any "reasonable" class table $\mathcal{T}$ that declares types $\mathbb{Z}$ and String would also declare one to exclude the other, so $\mathbb{Z} \lozenge \mathrm{String}$.

However, the Meet Rule still requires a third declaration in $\mathcal{D}(\mathit{simple})$ that is applicable to every type T if and only if simple_1 and simple_2 are applicable to T. Such T would necessarily be a subtype of both $\mathbb{Z}$ and String, but since these types exclude, no such T exists (in $\mathcal{T}$ or in any extension thereof). We thus augment every collection of overloaded function declarations $\mathcal{D}$ such that each $\mathcal{D}(f)$ includes an additional, implicit declaration $f_\perp \mathrm{BottomType}\!:\!\mathrm{Any}$. This declaration is trivially more specific than any declaration possibly written by a programmer, but it does not conflict with overloading safety since it is only applicable to $\mathrm{BottomType}$.

With $\mathit{simple}_\perp$ implicitly part of $\mathcal{D}(\mathit{simple})$, the two simple declarations *almost* satisfy the Meet Rule: the checker now must verify that $\mathit{dom}(\mathit{simple}_\perp)$, $\mathrm{BottomType}$, is equivalent to the computed meet, $\mathrm{String} \cap \mathbb{Z}$. Therefore we augment our judgment for the subtype relation with the rule necessary for constructing this equivalence:

$$\frac{S \lozenge T}{\Delta \vdash S \cap T <: \mathrm{BottomType}}$$

With these adjustments, $\mathcal{D}(\mathit{simple})$ is now a valid overloading.

6.3 Overloading Rule with Polymorphic Exclusion

Checking the declarations $\mathcal{D}(\mathit{tail})$ and $\mathcal{D}(\mathit{foo})$ is trickier. Certainly neither List nor $\mathrm{ArrayList}$ declares an exclusion of the other, so seemingly the exclusion relation does not help us check these declarations. A fundamental problem is that a type such as $\mathrm{BadList}$ might be a subtype of multiple instantiations of the type constructor List (in this case, a subtype of $\mathrm{List}[\![\mathbb{Z}]\!]$ directly and of $\mathrm{List}[\![\mathrm{String}]\!]$ via $\mathrm{ArrayList}[\![\mathrm{String}]\!]$). A type hierarchy in which some type extends multiple instantiations of the same type constructor is said to exhibit *multiple instantiation inheritance* [8].

We address this problem by imposing an additional restriction on well-formed class tables, which, in effect, adds additional power to the exclusion relation:

Polymorphic Exclusion Rule A type (other than $\mathrm{BottomType}$) must not be a subtype of multiple distinct instantiations of a type constructor.

Thus, distinct well-formed applications $C[\![\overline{T}]\!]$ and $C[\![\overline{U}]\!]$ of a type constructor C exclude each other; we use $\lozenge_\mathrm{p}$ for this relation. Furthermore, if M is a subtype of $C[\![\overline{T}]\!]$ and N is a subtype of $C[\![\overline{U}]\!]$, then M and N also exclude each other; we use $\lozenge_\mathrm{p}$ for this relation, and we add it to the other exclusion sub-relations whose union forms the exclusion relation over constructed types.

Polymorphic exclusion is easy to statically enforce, and practice suggests that it is not onerous: it is already in the Java™ programming language. Furthermore, there is good reason to enforce it: Kennedy and Pierce have shown that multiple instantiation inheritance is one of three conditions that lead to the undecidability of nominal subtyping with variance [8].

[6] For the sake of catching likely programming errors, the Fortress language requires that every K_i in a `comprises` clause for $C[\![\overline{T}]\!]$ be a subtype of $C[\![\overline{T}]\!]$, but allowing Any to appear in a `comprises` clause simplifies our presentation here.

With polymorphic exclusion in place, the type $\mathrm{BadList}$ is no longer well-formed in its class table. However, our subtype judgment still cannot find a type $W <: \mathbb{Z}$ to prove that:

$$X <: \mathrm{Any}, Y <: \mathbb{Z} \vdash \mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!] <: \mathrm{ArrayList}[\![W]\!]$$

which is necessary to satisfy the Meet Rule. All that is known about X and Y for this subtype check is that they are subtypes of Any and $\mathbb{Z}$, respectively. If we could somehow equate X and Y, transferring Y's tighter bound $\mathbb{Z}$ to X, then we could use $W = X$ to prove the assertion. Syntactically, the types X and Y do not provide enough information to prove the subtype assertion, but with additional assumptions about their structure (namely, that their extents are equal) we can indeed prove the assertion. Incorporating this kind of deduction into our subtyping judgment as described in Section 6.4 allows many more sets of overloaded function declarations such as *tail* and *foo* as valid overloadings.

6.4 Exclusion with Constraints

Constraints We now augment our subtyping judgment to include *constraints*. With constraint-based subtyping, we perform backward reasoning from the desired subtyping assertion to derive constraints on types under which the assertion can be proved; when those constraints are satisfied, the assertion is also satisfied. These constraints on types are first generated and gathered, and solved later when more information about the environment is known.

Our grammar for type constraints is defined as follows:

$$\begin{array}{rcl} \mathcal{C} & ::= & X <: S \\ & | & S <: X \\ & | & X \lozenge S \\ & | & X \not<: S \\ & | & S \not<: X \\ & | & X \not\lozenge S \\ & | & \mathcal{C} \wedge \mathcal{C} \\ & | & \mathcal{C} \vee \mathcal{C} \\ & | & \mathrm{false} \\ & | & \mathrm{true} \end{array}$$

A primitive constraint is either *positive* or *negative*. A positive primitive constraint of the form $X <: S$ specifies that a type variable X is a subtype of S, and likewise for $S <: X$. The positive primitive constraint $X \lozenge S$ specifies that a type variable X must exclude a type S. To generate constraints for exclusion, we also define negative primitive constraints: $X \not<: S$, $S \not<: X$, and $X \not\lozenge S$. A conjunction constraint $\mathcal{C}_1 \wedge \mathcal{C}_2$ is satisfied exactly when both $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfied, and a disjunction constraint $\mathcal{C}_1 \vee \mathcal{C}_2$ is satisfied exactly when one or both of $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfied. The constraint false is never satisfied, and the constraint true is always satisfied.

Constraint Generation We introduce the following syntactic judgments to generate these constraints:

$$\Delta \vdash S <: T \mid \mathcal{C} \qquad \Delta \vdash S \equiv T \mid \mathcal{C} \qquad \Delta \vdash S \lozenge T \mid \mathcal{C}$$

Each judgment indicates that, under assumptions Δ, the respective predicate can be proved if the constraint $\mathcal{C}$ is satisfied. An important point about these judgments is that the predicates $S <: T$, $S \equiv T$, and $S \lozenge T$ should be considered *inputs* to our algorithmic checker, while the constraint $\mathcal{C}$ should be considered its *output*.

We do not give the full semantics of constraint generation for subtyping. Instead, we refer the reader to the recent work by Smith and Cartwright [18] on which our system was based. Smith and Cartwright provide a sound and complete algorithm for generating constraints from the subtyping relation and an algorithm for normalizing constraints to simplify, for example, those involving contradictions or redundancies. We assume that constraints are implicitly simplified in this manner. The soundness of constraint generation entails that the predicate is a logical consequence of the constraint; the completeness of generation entails the opposite.

The constraint generation judgment for equivalence is defined entirely by the following rule:

$$\frac{\Delta \vdash S <: T \mid \mathcal{C} \qquad \Delta \vdash T <: S \mid \mathcal{C}'}{\Delta \vdash S \equiv T \mid \mathcal{C} \wedge \mathcal{C}'}$$

Equivalence constraint generation is complete and sound if and only if the constraint generation for $<:$ is.

With negative constraints, we define a new judgment for "not a subtype" by the rule:

$$\frac{\Delta \vdash S <: T \mid \mathcal{C}}{\Delta \vdash S \not<: T \mid \neg\mathcal{C}}$$

where $\neg\mathcal{C}$ is the negated constraint formed by applying De Morgan's laws on $\mathcal{C}$. Similarly, we can define "not equivalent" by the rule:

$$\frac{\Delta \vdash S \equiv T \mid \mathcal{C}}{\Delta \vdash S \not\equiv T \mid \neg\mathcal{C}}$$

These judgements are sound because constraint generation for $<:$ is complete (and complete because $<:$ is sound).

With these additional constraints and judgments, we can now define constraint generation for exclusion. Figure 1 presents our algorithm, which formalizes our original definition of exclusion in Section 6.1. As before, exclusion on constructed types is satisfied when any of the sub-relations from the previous section is satisfied. Each sub-relation checks the conditions described before by recursively generating and propagating constraints. Each sub-relation except $\triangleright_{\mathrm{c}}$ depends only on other constraint generation judgments, meaning the algorithmic checking terminates. The $C[\![\overline{T}]\!] \triangleright_{\mathrm{c}} D[\![\overline{U}]\!]$ predicate recursively checks exclusion on the comprised

Generating constraints for exclusion: $\Delta \vdash T \lozenge T \mid \mathcal{C}$

Symmetry

$$\frac{\Delta \vdash T \lozenge S \mid \mathcal{C}}{\Delta \vdash S \lozenge T \mid \mathcal{C}}$$

Structural rules

$$\frac{}{\Delta \vdash \text{BottomType} \lozenge T \mid \text{true}}$$

$$\frac{\Delta \vdash T <: \text{BottomType} \mid \mathcal{C}}{\Delta \vdash \text{Any} \lozenge T \mid \mathcal{C}}$$

$$\frac{|\overline{S}| \neq |\overline{T}|}{\Delta \vdash (\overline{S}) \lozenge (\overline{T}) \mid \text{true}}$$

$$\frac{|\overline{S}| = |\overline{T}| \qquad \overline{\Delta \vdash S \lozenge T \mid \mathcal{C}}}{\Delta \vdash (\overline{S}) \lozenge (\overline{T}) \mid \bigvee \mathcal{C}_i}$$

$$\frac{|\overline{T}| \neq 1}{\Delta \vdash (S \to R) \lozenge (\overline{T}) \mid \text{true}}$$

$$\frac{C \neq \text{Any} \qquad |\overline{T}| \neq 1}{\Delta \vdash C[\![\overline{S}]\!] \lozenge (\overline{T}) \mid \text{true}}$$

$$\frac{C \neq \text{Any}}{\Delta \vdash C[\![\overline{S}]\!] \lozenge (T \to U) \mid \text{true}}$$

$$\frac{\Delta \vdash S \lozenge U \mid \mathcal{C} \qquad \Delta \vdash T \lozenge U \mid \mathcal{C}' \qquad \Delta \vdash S \lozenge T \mid \mathcal{C}''}{\Delta \vdash S \cap T \lozenge U \mid \mathcal{C} \vee \mathcal{C}' \vee \mathcal{C}''}$$

$$\frac{}{\Delta \vdash (S \to T) \lozenge (U \to V) \mid \text{false}}$$

$$\frac{\Delta \vdash S \lozenge U \mid \mathcal{C} \qquad \Delta \vdash T \lozenge U \mid \mathcal{C}'}{\Delta \vdash S \cup T \lozenge U \mid \mathcal{C} \wedge \mathcal{C}'}$$

Type variables

$$\frac{}{\Delta \vdash X \lozenge T \mid X \lozenge T}$$

Constructed types

$$\frac{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\mathrm{e}} D[\![\overline{T}]\!] \mid \mathcal{C}_e \quad \Delta \vdash C[\![\overline{S}]\!] \lozenge_{\mathrm{c}} D[\![\overline{T}]\!] \mid \mathcal{C}_c \quad \Delta \vdash C[\![\overline{S}]\!] \lozenge_{\mathrm{o}} D[\![\overline{T}]\!] \mid \mathcal{C}_o \quad \Delta \vdash C[\![\overline{S}]\!] \lozenge_{\mathrm{p}} D[\![\overline{T}]\!] \mid \mathcal{C}_p}{\Delta \vdash C[\![\overline{S}]\!] \lozenge D[\![\overline{T}]\!] \mid \mathcal{C}_e \vee \mathcal{C}_c \vee \mathcal{C}_o \vee \mathcal{C}_p}$$

$$\frac{\Delta \vdash C[\![\overline{S}]\!] \rhd_x D[\![\overline{T}]\!] \mid \mathcal{C} \qquad \Delta \vdash D[\![\overline{T}]\!] \rhd_x C[\![\overline{S}]\!] \mid \mathcal{C}'}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_x D[\![\overline{T}]\!] \mid \mathcal{C} \vee \mathcal{C}'} \quad \text{where } x \in \{\mathrm{e}, \mathrm{c}, \mathrm{o}\}$$

$$\frac{}{\Delta \vdash C[\![\overline{S}]\!] \rhd_x C[\![\overline{T}]\!] \mid \text{false}} \quad \text{where } x \in \{\mathrm{e}, \mathrm{c}, \mathrm{o}\}$$

$$\frac{C \neq D \qquad A = \mathit{ancestors}(C[\![\overline{S}]\!]) \qquad \forall U \in \left(\bigcup_{N \in A} N.\mathit{excludes}\right). \quad \Delta \vdash D[\![\overline{T}]\!] <: U \mid \mathcal{C}_U}{\Delta \vdash C[\![\overline{S}]\!] \rhd_{\mathrm{e}} D[\![\overline{T}]\!] \mid \bigvee \mathcal{C}_U}$$

$$\frac{C \neq D \qquad \forall U \in C[\![\overline{S}]\!].\mathit{comprises}. \quad \Delta \vdash U \lozenge D[\![\overline{T}]\!] \mid \mathcal{C}_U}{\Delta \vdash C[\![\overline{S}]\!] \rhd_{\mathrm{c}} D[\![\overline{T}]\!] \mid \mathcal{C} \wedge \bigwedge \mathcal{C}_U}$$

$$\frac{C \text{ does not have a } \texttt{comprises} \text{ clause}}{\Delta \vdash C[\![\overline{S}]\!] \rhd_{\mathrm{c}} D[\![\overline{T}]\!] \mid \text{false}}$$

$$\frac{\texttt{object}\, C \quad C \neq D \quad \Delta \vdash C[\![\overline{S}]\!] \not<: D[\![\overline{T}]\!] \mid \mathcal{C}}{\Delta \vdash C[\![\overline{S}]\!] \rhd_{\mathrm{o}} D[\![\overline{T}]\!] \mid \mathcal{C}}$$

$$\frac{\neg(\texttt{object}\, C)}{\Delta \vdash C[\![\overline{S}]\!] \rhd_{\mathrm{o}} D[\![\overline{T}]\!] \mid \text{false}}$$

$$\frac{\forall U \in \mathit{ancestors}(C[\![\overline{S}]\!]). \quad \forall V \in \mathit{ancestors}(D[\![\overline{T}]\!]). \quad \Delta \vdash U \lozenge_{\mathrm{p}} V \mid \mathcal{C}_{U,V}}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\mathrm{p}} D[\![\overline{T}]\!] \mid \bigvee \mathcal{C}_{U,V}}$$

$$\frac{\overline{\Delta \vdash S \not\equiv T \mid \mathcal{C}}}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\mathrm{p}} C[\![\overline{T}]\!] \mid \bigvee \mathcal{C}_i}$$

$$\frac{C \neq D}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\mathrm{p}} D[\![\overline{T}]\!] \mid \text{false}}$$

Figure 1. Generating constraints for exclusion

types of $C[\![\overline{T}]\!]$, but due to the acyclic nature of the class table, this process will also terminate. Thus, the algorithm is complete and sound if $<:$ is.

For subtyping with constraints, we essentially preserve the semantics of [18], but that system lacks our notion of exclusion. Since we need our subtyping to take advantage of exclusion, we must add an additional rule to the judgment:

$$\frac{\Delta \vdash S \lozenge T \,|\, \mathcal{C} \qquad \Delta \vdash S <: U \,|\, \mathcal{C}' \qquad \Delta \vdash T <: U \,|\, \mathcal{C}''}{\Delta \vdash S \cap T <: U \,|\, \mathcal{C} \vee \mathcal{C}' \vee \mathcal{C}''}$$

This new rule states three possibilities for proving that the intersection $S \cap T$ is a subtype of U: S and T exclude (which means their intersection is a subtype of $\mathrm{BottomType}$), S is a subtype of U, or T is a subtype of U. If any of the constraints needed for these three judgments is satisfied, then $S \cap T$ is a subtype of U. Adding this rule should not affect completeness or soundness of constraint generation for $<:$ (since constraint generation for $\lozenge$ is complete and sound if constraint generation for $<:$ is).

Constraint Solving Solving constraints needs to be sound but not necessarily complete.[7] Here is a simple algorithm which is sound but not complete. (We assume that all constraints are kept in disjunctive normal form.)

1. Take $\mathcal{C}$ which is a union of intersection constraints and try to solve each conjunct at a time (take the first one that succeeds).
2. Split the conjunct into positive and negative parts.
3. Deduce a set of equivalences from the positive part.
4. Solve the equivalences using unification (with subtyping) to get a substitution ϕ.
5. Check whether applying ϕ to $\mathcal{C}$ reduces $\mathcal{C}$ to the trivial constraint true.
6. If so return $\mathrm{Some}(\phi)$, otherwise return None.

To get better completeness properties, one can iterate constraint solving if $\phi(\mathcal{C})$ is not false. However, this might converge to a fixed point instead of terminating because the constraints that are negative or that do not imply an equivalence are never used to generate ϕ.

7. Overloading Rules Checking

This section describes how to mechanically check the three overloading rules described in Section 3.1. Because the rules use the notions of applicability and specificity for generic functions, we first formalize them which in turn need formalization of the "arrow type of f" for a generic function declaration f.

For a generic function declaration $f[\![\Delta]\!]\, S{:}T$, its arrow type (written *arrow*(f)) is the *universal type* $\forall[\![\Delta]\!]S \to T$. A universal type binds type parameter declarations over some type and can be instantiated by any types valid for those type parameters. We write $\forall[\![\overline{X <: \{N\}}]\!]T$ to quantify each type variable X_i with bounds $\{\overline{N_i}\}$ over the type T, and we use the metavariable σ to range over universal types. The metalevel function `FV` maps a type to all free variables contained in that type.

Recall that a generic function declaration $f[\![\Delta]\!]\, S{:}T$ is applicable to a type U if and only if some instance of f is applicable to U. A monomorphic function declaration $f\ S'{:}T'$ is applicable to a type U if and only if $U <:$ *dom*(f), where *dom*(f) is the domain S' of $S' \to T'$, the arrow type of f. This existential quantification over possible instantiations of the domain directly corresponds to an existentially quantified type as in [3]. An *existential type* $\exists[\![\overline{X <: \{N\}}]\!]T$ also binds type parameter declarations over a type, but unlike a universal type, it cannot be instantiated; instead, it represents some hidden type instantiation $\overline{W}$ and the corresponding instantiated type $[\overline{W}/\overline{X}]T$. Therefore, we say that the domain of the universal arrow type for f, $\forall[\![\Delta]\!]S \to T$ (again written *dom*(f) as an abuse of notation), is the existential type $\exists[\![\Delta]\!]S$. We use the metavariable δ to range over existential types. Note that we abbreviate both the universal type $\forall[\![]\!]T$ and the existential type $\exists[\![]\!]T$ as simply T when the meaning is clear from context. Figure 2 presents the formal definition of applicability for a generic function f to a type T, $f \ni T$.

A generic function declaration is *more specific* than another generic function declaration if and only if the domain of the former is a subtype of the latter as presented in Figure 2. Figure 2 also presents the subtype relation on existential types as originally given by Mitchell [13]. Roughly, an existential type $\delta_1 = \exists[\![\Delta_1]\!]T_1$ is a subtype of another existential type $\delta_2 = \exists[\![\Delta_2]\!]T_2$ (written $\delta_1 \leq \delta_2$) if T_2 can be instantiated to a supertype of T_1 in the environment Δ_1.

Now, we can mechanically check the No Duplicates Rule by mechanically determining if two declarations are equally specific. To check whether one declaration is more specific than another, we check whether the domain of the former is a subtype of the latter. Checking whether one existential type is a subtype of another is described in Figure 2.

To check the Meet Rule, for every pair of declarations $f_1, f_2 \in \mathcal{D}(f)$, we must find a function declaration $f_0 \in \mathcal{D}(f)$ that is applicable to a type T if and only if both f_1 and f_2 are applicable to T. In other words, we need to find f_0 that is equivalent under specificity to the meet of f_1 and f_2. To mechanically find the meet of two generic function declarations, we define the *computed meet* of f_1 and f_2 as a declaration $f_\wedge$, not necessarily in $\mathcal{D}(f)$, such

[7] A failure in solving constraints may cause us to reject some overloadings that were actually safe, but not to allow overloadings that are unsafe.

Universal subtyping: $\Delta \vdash \sigma \leq \sigma$

$$\frac{\mathit{fresh}(\overline{Z}) \qquad \Delta' = \Delta, \overline{Z <: \{\overline{N}\}} \qquad \Delta' \vdash [\overline{V}/\overline{X}]T <: [\overline{Z}/\overline{Y}]U \qquad \forall i\,.\,\Delta' \vdash V_i <: [\overline{V}/\overline{X}]\{\overline{M_i}\}}{\Delta \vdash \forall[\![\overline{X <: \{\overline{M}\}}]\!]T \leq \forall[\![\overline{Y <: \{\overline{N}\}}]\!]U}$$

TODO: Perhaps $\overline{Z}$ should be defined in a "where" side condition.

$$\sigma_1 \leq \sigma_2 \quad \stackrel{\text{def}}{=} \quad \emptyset \vdash \sigma_1 \leq \sigma_2$$

Existential subtyping: $\Delta \vdash \delta \leq \delta$

$$\frac{\mathit{fresh}(\overline{Z}) \qquad \Delta' = \Delta, \overline{Z <: \{\overline{N}\}} \qquad \Delta' \vdash [\overline{Z}/\overline{X}]T <: [\overline{V}/\overline{Y}]U \qquad \forall i\,.\,\Delta' \vdash V_i <: [\overline{V}/\overline{Y}]\{\overline{N_i}\}}{\Delta \vdash \exists[\![\overline{X <: \{\overline{M}\}}]\!]T \leq \exists[\![\overline{Y <: \{\overline{N}\}}]\!]U}$$

$$\delta_1 \leq \delta_2 \quad \stackrel{\text{def}}{=} \quad \emptyset \vdash \delta_1 \leq \delta_2$$

Applicability: $\Delta \vdash \delta \ni T$ $\quad f \ni T$

$$\frac{\Delta \vdash \exists[\![]\!]T \leq \delta}{\Delta \vdash \delta \ni T}$$

$$\delta \ni T \quad \stackrel{\text{def}}{=} \quad \emptyset \vdash \delta \ni T$$
$$f \ni T \quad \stackrel{\text{def}}{=} \quad \emptyset \vdash \mathit{dom}(f) \ni T$$

Specificity: $\Delta \vdash f \preceq f$

$$\frac{\Delta \vdash \mathit{dom}(f_1) \leq \mathit{dom}(f_2)}{\Delta \vdash f_1 \preceq f_2}$$

$$f_1 \preceq f_2 \quad \stackrel{\text{def}}{=} \quad \emptyset \vdash f_1 \preceq f_2$$

Existential meet: $\delta_1 \wedge \delta_2$

$$\left(\exists[\![\overline{X <: \{\overline{M}\}}]\!]T\right) \wedge \left(\exists[\![\overline{Y <: \{\overline{N}\}}]\!]U\right) \quad \stackrel{\text{def}}{=} \quad \exists[\![\overline{X' <: \{\overline{M}\}}, \overline{Y' <: \{\overline{N}\}}]\!]([\overline{X'}/\overline{X}]T \cap [\overline{Y'}/\overline{Y}]U)$$
where $\mathit{fresh}(\overline{X'}, \overline{Y'})$

Figure 2. Subtyping on universal and existential types, applicability and specificity on generic function declarations, and meet of existential types

that $\mathit{dom}(f_\wedge) \equiv \mathit{dom}(f_1) \wedge \mathit{dom}(f_2)$[8]:

$$\begin{aligned} \mathit{dom}(f_0) &\leq \mathit{dom}(f_1) \wedge \mathit{dom}(f_2) \\ \mathit{dom}(f_1) \wedge \mathit{dom}(f_2) &\leq \mathit{dom}(f_0) \end{aligned}$$

Figure 2 defines the meet of two existential types $\delta_1 \wedge \delta_2$, which may require alpha renaming on the existential types' parameters. The following lemma shows that the definition of the meet of two existential types is correct:

Lemma 3 $\delta_1 \wedge \delta_2$ is the meet of δ_1 and δ_2 under $\leq$.

Proof Sketch: That $\delta_1 \wedge \delta_2$ is a subtype of both δ_1 and δ_2 is obvious. For any δ_0, if $\overline{U}$ and $\overline{V}$ are instantiations that prove δ_0 is a subtype of δ_1 and δ_2, respectively, then we can use $\overline{U}, \overline{V}$ to prove that δ_0 is a subtype of $\delta_1 \wedge \delta_2$. □

We can check the Return Type Rule using the subtype relation on universal types. Figure 2 presents the subtype relation on universal types, again, as originally given by Mitchell [13]. Roughly, a universal type $\sigma_1 = \forall[\![\Delta_1]\!]T_1$ is a subtype of another universal type $\sigma_2 = \forall[\![\Delta_2]\!]T_2$ (written $\sigma_1 \leq \sigma_2$) if T_1 can be instantiated to a subtype of T_2 in the environment Δ_2.

[8] Note that the computed meet, as defined, is not actually unique since the return type is unspecified. By an abuse of notation, we refer to "the" computed meet to mean any such computed meet.

The generic declarations $f_1 = f[\![\Delta_1]\!]\, S_1 : T_1$ and $f_2 = f[\![\Delta_2]\!]\, S_2 : T_2$ with $f_1 \preceq f_2$ satisfy the Return Type Rule whenever:

$$\forall[\![\Delta_1]\!]S_1 \to T_1 \;\leq\; \forall[\![\Delta_1, \Delta_2]\!](S_1 \cap S_2) \to T_2 \qquad (*)$$

To see this, suppose W is a type to which f_1 is applicable and the instance $f_2'\, S_2' : T_2'$ of f_2 is also applicable to W. Let $f_\wedge = f[\![\Delta_1, \Delta_2]\!]\, S_1 \cap S_2 : T_2$ be the computed meet of f_1 and f_2; clearly, there is an instance $f_\wedge'\, U' : V'$ of $f_\wedge$ that is applicable to W with $V' = T_2'$. If $\mathit{arrow}(f_1) \leq \mathit{arrow}(f_\wedge)$, then there must be an instantiation $f_1'\, S_1' : T_1'$ of f_1 with $U' <: S_1'$ and $T_1' <: V'$. Moreover, since $V' = T_2'$, such an f_1' would satisfy the Return Type Rule for f_1 and f_2. Finally, observe that $\mathit{arrow}(f_1) \leq \mathit{arrow}(f_\wedge)$ is identical to the condition $(*)$, and so the verification of $(*)$ implies the verification of the Return Type Rule.

7.1 Incorporating Constraints into Subtyping

We can recover the usual subtyping judgment and define the exclusion judgment as follows:

$$\frac{\Delta \vdash S <: T \mid \text{true}}{\Delta \vdash S <: T} \qquad \frac{\Delta \vdash S \lozenge T \mid \text{true}}{\Delta \vdash S \lozenge T}$$

which state that, under assumptions Δ, if the predicate generates the constraint true (or some constraint that simplifies to true), then that predicate is proved.

Existential reduction: $\vdash \delta \xrightarrow{\equiv} \delta\,,\ \phi$

$$\frac{\Delta \vdash T \equiv \mathrm{BottomType} \mid \mathsf{true}}{\vdash \exists[\![\Delta]\!]T \xrightarrow{\equiv} \mathrm{BottomType}\,,\ []}$$

$$\frac{\Delta \vdash T \not\equiv \mathrm{BottomType} \mid \mathcal{C} \qquad \Delta \vdash \mathit{solve}(\mathcal{C}) = \phi \qquad \phi[\Delta] = \Delta'}{\vdash \exists[\![\Delta]\!]T \xrightarrow{\equiv} \exists[\![\Delta']\!]\phi(T)\,,\ \phi}$$

Bounds substitution: $\phi[\Delta] = \Delta$

$$\frac{\Delta = \overline{X <: \{\overline{M}\}} \qquad \phi(\overline{X}) = \overline{Y}, \overline{T} \qquad \overline{\overline{N}} = \overline{\phi^{-1}[Y, \Delta]} \qquad \forall i.\quad \overline{Y <: \{\overline{N}\}} \vdash \phi(X_i) <: \overline{\phi(M_i)} \mid \mathsf{true}}{\phi[\Delta] = \overline{Y <: \{\overline{N}\}}}$$

Bounds transfer: $\phi^{-1}[X, \Delta] = \overline{T}$

$$\frac{\{\overline{T}\} = \{\phi(\Delta(X)) \mid X \in \mathit{dom}(\Delta),\ \phi(X) = Y\}}{\phi^{-1}[Y, \Delta] = \mathit{conjuncts}(\bigcap \overline{T})}$$

$$\mathit{conjuncts}(T) \stackrel{\text{def}}{=} \begin{cases} \mathit{conjuncts}(T_1), \mathit{conjuncts}(T_2) & \text{if } T = T_1 \cap T_2 \\ T & \text{otherwise} \end{cases}$$

Figure 3. Reduction of existential types

To allow more overloaded functions such as tail and foo as valid overloadings, we adjust the subtype relation to take into account the relationships between type variables:

$$\frac{\vdash \delta \xrightarrow{\equiv} \delta' \qquad \Delta \vdash \delta' \leq T}{\Delta \vdash \delta \leq T}$$

$$\frac{\vdash \exists[\![\Delta_1]\!]T \xrightarrow{\equiv} \delta'\,,\ \phi \qquad \Delta \vdash S \leq \forall[\![\phi[\Delta_1]]\!](\phi(T) \to \phi(U))}{\Delta \vdash S \leq \forall[\![\Delta_1]\!]T \to U}$$

A reduction judgment on existential types $\vdash \delta \xrightarrow{\equiv} \delta', \phi$ defined in Figure 3 reduces $\delta = \exists[\![\Delta]\!]T$ to $\delta' = \exists[\![\Delta']\!]T'$ with the substitution ϕ under the assumption that T is not equivalent to $\mathrm{BottomType}$; or, if T is equivalent to $\mathrm{BottomType}$, the reduced existential δ' is $\mathrm{BottomType}$. When the substitution is unnecessary we omit it. The solve operation is like that defined in Smith and Cartwright [18]. In fact, an existential type δ reduces to δ' such that $T \ni \delta$ if and only if $T \ni \delta'$; therefore, we have $\delta' \leq \delta$. Reducing an existential type in this fashion involves the same kind of type analysis required for type checking generalized algebraic data types [15, 17].

As an example, consider the following reduction:

$$\vdash \exists[\![X <: \mathrm{Any}, Y <: \mathbb{Z}]\!]\big(\mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!]\big) \xrightarrow{\equiv} \exists[\![W <: \mathbb{Z}]\!]\mathrm{ArrayList}[\![W]\!]$$

with substitution $[W/X,\ W/Y]$. We first check under what constraints $\mathcal{C}$ the intersection $\mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!]$ is not equivalent to $\mathrm{BottomType}$: with polymorphic exclusion (specifically, the absence of multiple instantiation inheritance) we know that $\mathrm{ArrayList}[\![X]\!]$ excludes $\mathrm{List}[\![Y]\!]$, which makes their intersection equivalent to $\mathrm{BottomType}$, unless $X \equiv Y$ is true. Solving the constraint $X \equiv Y$ yields a type substitution like $\phi = [W/X, W/Y]$. The judgment $\phi[\Delta] = \Delta'$ lets us construct reduced bounds from ϕ and the original bounds Δ. To do this, we first partition $\phi(\overline{X})$ into a list of type variables $\overline{Y}$ and a list of other types $\overline{T}$. In our example, $\phi(X, Y) = W$ gets partitioned into W and $\emptyset$. Then we need to construct a new bound $\phi^{-1}[Y_i, \Delta]$ for each Y_i in $\overline{Y}$ by conjoining the bounds for every type variable in $\phi^{-1}(Y)$. In our example, X and Y map to W, so the bounds for W are $\{\mathrm{Any}, \mathbb{Z}\}$, which we take as the new bounds environment Δ'. We must ensure that the substitution does not produce invalid bounds, so we check $\Delta' \vdash \phi(X) <: \overline{\phi(M)}$. In our example, $W <: \{\mathbb{Z}, \mathrm{Any}\}$ easily proves that $W <: \mathbb{Z}$ and $W <: \mathrm{Any}$. With Δ' the reduced existential type is simply $\exists[\![\Delta']\!]\phi(T)$, where T is the constituent type of the original existential. In our example, the final, reduced existential type is $\exists[\![W <: \mathbb{Z}]\!]\mathrm{ArrayList}[\![W]\!]$. Once we have augmented the subtyping relation with existential reduction, we can finally check that the declarations $\mathcal{D}(\mathit{foo})$ from Section 5 satisfy the Meet Rule.

Similiarly, to check that the declarations $\mathcal{D}(\mathit{tail})$ from Section 5 satisfy the Return Type Rule, we need to show the following:

$$\vdash \forall[\![X <: \mathrm{Any}]\!]\mathrm{ArrayList}[\![X]\!] \to \mathrm{ArrayList}[\![X]\!] \leq \forall[\![X <: \mathrm{Any}, Y <: \mathrm{Any}]\!]\big(\mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!]\big) \to \mathrm{List}[\![Y]\!]$$

By the adjusted subtype relation in this section and the rules in Figure 3, we can show the following:

$$\vdash \exists[\![X <: \mathrm{Any}, Y <: \mathrm{Any}]\!]\mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!] \xrightarrow{\equiv} \exists[\![W <: \mathrm{Any}]\!]\mathrm{ArrayList}[\![W]\!]$$

with substitution $[W/X, W/Y]$. Because the substitution satisfies the following:

$$\vdash \forall[\![X <: \mathrm{Any}]\!]\mathrm{ArrayList}[\![X]\!] \to \mathrm{ArrayList}[\![X]\!] \leq \forall[\![X <: \mathrm{Any}]\!]\mathrm{ArrayList}[\![X]\!] \to \mathrm{List}[\![X]\!]$$

we can verify that the declarations $\mathcal{D}(\mathit{tail})$ satisfy the Return Type Rule.

8. Discussion

8.1 Type Inference

Before describing a system for ensuring progress and preservation, it is important to discuss some implications of these conditions on type inference in a programming language. The application of a function declaration to a type requires instantiation of the type parameters of the declaration. In most programming languages with parametric polymorphism, a type inference mechanism automatically instantiates type parameters based on the types of the arguments and the enclosing context. But note that our progress and preservation conditions do not require that the type parameters of the function declaration that is (dynamically) most specific of those applicable to the ilks of the argument values are the same as the type parameters of the function declaration that is (statically) most specific of those applicable to the static types of the argument expressions. Thus, the results of static type inference do not tell us how to instantiate the type parameters of a most specific function declaration at run time. In the Fortress programming language, type inference is performed statically, and the results of that inference are passed to the run-time system to ensure that run-time type inference at a function call is sound. The rules for overloaded function declarations introduced in Section 3 ensure that the declaration of the dynamically most applicable function declaration, when instantiated with whatever we infer at run time, is more specific than the declaration of the statically most applicable function declaration, instantiated with what was inferred at compile time.

Aside from this caveat, our system for checking overloaded declarations is largely independent of how a specific type inference engine would choose instantiations[9]. Thus we do not discuss the specific features of a type inference system further in this paper.

8.2 Modularity

To demonstrate the modularity of our design, we present a lightweight modeling of program modules, and show how applying our rules to each module separately suffices to guarantee the soundness of the entire program. In our model, a program is a *module*, which may be either *simple* or *compound*. A *simple module* consists of (i) a class table and (ii) a collection of function declarations. That is, a simple module is just a program as described in the rest of this paper. It is well-formed if it satisfies the well-formedness conditions of a whole program, as described in previous sections.

A *compound module* combines multiple modules, possibly renaming members (i.e., classes and functions) of its constituents. More precisely, a compound module is a collection of *filters*, where a filter consists of a module and a complete mapping from names of members of the module to names. The name of a member that is not renamed is simply mapped to itself.

The semantics of a compound module is the semantics of the simple module that results from recursively *flattening* the compound module as follows:

- Flattening a simple module simply yields the same module.
- Flattening a compound module C consisting of filters (module/mapping pairs) $(c_1, m_1), \ldots, (c_N, m_N)$ yields a simple module whose class table and collection of function declarations is the concatenation of the class tables and collections of function declarations of $s_1, \ldots, s_N$, where s_i is the simple module resulting from first flattening c_i and then renaming all members of the resulting simple module according to the mapping m_i.

A compound module is well-formed if its flattened version is well-formed. This requirement implies that the type hierarchies in each constituent component are consistent with the type hierarchy in the flattened version.

We can now use this model of modularity to see that we can separately compile and combine modules.

First consider the case of a collection of modules with no overlapping function names such that each module has been checked separately to ensure that the overloaded functions within them satisfy the overloading rules. Because the type hierarchies of each constituent of a compound module must be consistent with that of the compound module, all overloaded functions in the resulting compound module also obey the overloading rules.

Now consider the case of a collection of separately checked modules with some overlapping function names. When overloaded functions from separate modules are combined, there are three rules that might be violated by the resulting overloaded definitions: (1) the Meet Rule, (2) the No Duplicates Rule, (3) the Return Type Rule. If the Meet Rule is violated, the programmer need only define yet another module to combine that defines the missing meets of the various overloaded definitions. If the No Duplicates Rule or the Return Type Rule is violated, the programmer can fix the problem by renaming functions from one or more combined components to avoid the clash; the programmer can then define another component with more overloadings of the same function name that dispatch to the various renamed functions in the manner the programmer intends.

Consider the following example:[10] Suppose we have a type Number in module A, with a function:

$$add : (\mathrm{Number}, \mathrm{Number}) \to \mathrm{Number}$$

Suppose we have the type and function:

$$\mathrm{BigNum} <: \mathrm{Number}$$
$$add : (\mathrm{BigNum}, \mathrm{BigNum}) \to \mathrm{BigNum}$$

[9] Type inference manifests itself as the choice of instantiation of type variables in existential and universal subtyping; specifically, $\overline{V}$ in the inference rules for subtyping in Figure 2. Mitchell [13] first showed how type inference interacts with polymorphic subtyping.

[10] Suggested by an anonymous reviewer of a previous version of this paper.

in module B and the type and function:

$$\text{Rational} <: \text{Number}$$
$$add : (\text{Rational}, \text{Rational}) \to \text{Rational}$$

in module C.

Each of modules B and C satisfy the No Duplicates and Meet rules. Now, suppose we define two compound modules D and E, each of which combines modules B and C without renaming add. In each of D and E, we have an ambiguity in dispatching calls to add with types $(\text{BigNum}, \text{Rational})$ or $(\text{Rational}, \text{BigNum})$. Our rules require adding two declarations in each of D and E to resolve these ambiguities.

Now let us suppose we wish to combine D and E into a compound component F. Without renaming, this combination would violate the No Duplicates Rule; each of D and E has an implementation of $add(\text{Bignum}, \text{Rational})$. To resolve this conflict, the program can rename add from both D and E, and define a new add in F. This new definition could dispatch to either of the renamed functions from D or E, or it could do something entirely different, depending on the programmer's intent.

9. Related Work

9.1 Overloading and dynamic dispatch.

TODO: Add discussion of Castagna et al. here?

Primarily, our system strictly extends our previous effort [2] with parametric polymorphism; all previous properties and results remain intact. The inclusion of parametric functions and types represents a shift in the research literature on overloading and multiple dynamic dispatch.

Millstein and Chambers [11, 12] devised the language Dubious to study overloaded functions with symmetric multiple dynamic dispatch (*multimethods*), and with Clifton and Leavens they developed MultiJava [5], an extension of Java with Dubious' semantics for multimethods. In [9], Lee and Chambers presented F(EML), a language with classes, symmetric multiple dispatch, and parameterized modules, but without parametricity at the level of functions or types. None of these systems support polymorphic functions or types. F(EML)'s parameterized modules (*functors*) constitute a form of parametricity but they cannot be implicitly applied; the functions defined therein cannot be *overloaded* with those defined in other functors. For a more detailed comparison of modularity and dispatch between our system and these, we refer to the related work section of our previous paper [2].

Overloading and multiple dispatch in the context of polymorphism has previously been studied by Bourdoncle and Merz [3]. Their system, $\text{ML}_{\leq}$, integrates parametric polymorphism, class-based object orientation, and multimethods, but lacks multiple inheritance. Each multimethod (overloaded set) requires a unique specification (principal type), which prevents overloaded functions defined on disjoint domains; and link-time checks are performed to ensure that multimethods are fully implemented across modules. On the other hand, $\text{ML}_{\leq}$ allows variance annotations on type constructors—something we attribute to future work.

Litvinov [10] developed a type system for the Cecil language, which supports bounded parametric polymorphism and multimethods. Because Cecil has a type-erasure semantics, statically checked parametric polymorphism has no effect on run-time dispatch.

9.2 Type classes.

Wadler and Blott [19] introduced *type class* as a means to specify and implement overloaded functions like equality and arithmetic operators in Haskell. Other authors have translated type classes to languages besides Haskell [6, 16, 20]. Type classes encapsulate overloaded function declarations, with separate *instances* that define the behavior of those functions (called *class methods*) for any particular type schema. Parametric polymorphism is then augmented to express type class constraints, providing a way to quantify a type variable — and thus a function definition — over all types that instantiate the type class.

In systems with type classes, overloaded functions must be contained in some type class, and their signatures must vary in exactly the same structural position. This uniformity is necessary for an overloaded function call to admit a principal type; with a principal type for some function call's context, the type checker can determine the constraints under which a correct overloaded definition will be found. Because of this requirement, type classes are ill-suited for fixed, *ad hoc* sets of overloaded functions like:

$$println() : () = println(\text{“”})$$
$$println(s : \text{String}) : () = \ldots$$

or functions lacking uniform variance in the domain and range[11] like:

$$bar(x : \mathbb{Z}) : \text{Boolean} = (x = 0)$$
$$bar(x : \text{Boolean}) : \mathbb{Z} = \texttt{if } x \texttt{ then } 1 \texttt{ else } 2 \texttt{ end}$$
$$bar(x : \text{String}) : \text{String} = x$$

With type classes one can write overloaded functions with identical domain types. Such behavior is consistent with the *static*, *type-based* dispatch of Haskell, but it would lead to irreconcilable ambiguity in the *dynamic*, *value-based* dispatch of our system.

A broader interpretation of Wadler and Blott's [19] sees type classes as program abstractions that quotient the space of ad-hoc polymorphism into the much smaller space of class methods. Indeed, Wadler and Blott's title suggests that the unrestricted space of ad-hoc polymorphism should be tamed, whereas our work embraces the possible expressivity

[11] With the *multi-parameter type class* extension, one could define functions as these. A reference to the method `bar`, however, would require an explicit type annotation like `:: Int -> Bool` to apply to an `Int`.

achieved from mixing ad-hoc and parametric polymorphism by specifying the requisites for determinism and type safety.

10. Conclusion and Discussion

We have shown how to statically ensure safety of overloaded, polymorphic functions while imposing relatively minimal restrictions solely on function definition sites. We provide rules on definitions that can be checked modularly, irrespective of call sites, and we show how to mechanically verify that a program satisfies these rules. The type analysis required for implementing these checks involves subtyping on universal and existential types, which adds complexity not required for similar checks on monomorphic functions. We have defined an object-oriented language to explain our system of static checks, and we have implemented them as part of the open-source Fortress compiler [1].

Further, we show that in order to check many "natural" overloaded functions with our system in the context of a generic, object-oriented language, richer type relations must be available to programmers—the subtyping relation prevalent among such languages does not afford enough type analysis alone. We have thus introduced an explicit, nominal exclusion relation to check safety of more interesting overloaded functions.

Variance annotations have proven to be a convenient and expressive addition to languages based on nominal subtyping [3, 8, 14]. They add additional complexity to polymorphic exclusion checking, so we leave them to future work.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0, March 2008.

[2] Eric Allen, J. J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele Jr. Modular multiple dispatch with multiple inheritance. In *SAC '07: Proceedings of the 2007 ACM Symposium on Applied Computing*, pages 1117–1121, New York, NY, USA, 2007. ACM.

[3] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 302–315, New York, NY, USA, 1997. ACM.

[4] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[5] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Trans. Program. Lang. Syst.*, 28(3):517–575, 2006.

[6] Derek Dreyer, Robert Harper, and Manuel M. T. Chakravarty. Modular type classes. In *POPL '07: Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 63–70, New York, NY, USA, 2007. ACM.

[7] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[8] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance, September 2006. FOOL-WOOD '07.

[9] Keunwoo Lee and Craig Chambers. Parameterized modules for classes and extensible functions. In *ECOOP '06 : Proceedings of the 20th European Conference on Object-Oriented Programming*, pages 353–378. Springer-Verlag, 2006.

[10] Vassily Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 388–411. ACM Press, 1998.

[11] Todd Millstein and Craig Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, 2002.

[12] Todd David Millstein. *Reconciling software extensibility with modular program reasoning*. PhD thesis, University of Washington, 2003.

[13] John C. Mitchell. Polymorphic type inference and containment. *Information and Computation*, 76(2-3):211–249, 1988.

[14] Martin Odersky. *The Scala Language Specification, Version 2.7*. EPFL Lausanne, Switzerland, 2009.

[15] Tom Schrijvers, Simon Peyton Jones, Martin Sulzmann, and Dimitrios Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352, New York, NY, USA, 2009. ACM.

[16] Jeremy G. Siek. *A language for generic programming*. PhD thesis, Indiana University, Indianapolis, IN, USA, 2005.

[17] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1):1, 2007.

[18] Daniel Smith and Robert Cartwright. Java type inference is broken: can we fix it? In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 505–524, New York, NY, USA, 2008. ACM.

[19] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76, New York, NY, USA, 1989. ACM.

[20] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized interfaces for Java. In *ECOOP 2007, Proceedings. LNCS, Springer-Verlag (2007) 25*, pages 347–372. Springer-Verlag, 2007.