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 that ensure type safety 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 additionally 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 also show how to check our rules for ensuring the safety of overloaded functions. In particular, we reduce the problem of handling parametric polymorphism to one of determining subtyping relationships among 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

OOPSLA '11 October 22–27, 2011, Portland, Oregon, USA.

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 at run time based on the "run-time types"—we use the term *ilks*—of the arguments, using the most specific definition that is applicable to arguments having those particular ilks. 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 ilks 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 a level of expressivity that closely models standard mathematical notation. In particular, mathematical operators such as $+$ and $\leq$ and $\cup$ and especially $\cdot$ and $\times$ have different definitions depending on the "types" (or even the number) of their operands; in a language with multiple dispatch, it is natural to define these operators as overloaded functions. Similarly, many operations on collections such as $append$ and zip have different definitions depending on the ilks of two or more arguments.

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

$$f(a\colon \mathrm{Object}, b\colon \mathbb{Z})\colon \mathbb{Z} = 1$$
$$f(a\colon \mathbb{Z}, b\colon \mathrm{Object})\colon \mathbb{Z} = 2$$

To which of these definitions ought we dispatch when f is called with two arguments of ilk $\mathbb{Z}$? (We assume that $\mathbb{Z}$ is a subtype of Object, written $\mathbb{Z} <: \mathrm{Object}$.)

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 domain type[1] of one is a subtype of the domain 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 domain types of the two definitions have a common lower bound (i.e., a common nontrivial[2] subtype), there is a unique definition for the same function whose domain type is the greatest lower bound of the domain types of the two definitions. Thus, to satisfy the latter property for the example above, the programmer must provide a third definition, such as:

$$f(a\colon \mathbb{Z}, b\colon \mathbb{Z})\colon \mathbb{Z} = 3$$

We call this latter property the *Meet Rule* because it is equivalent to requiring that the definitions for each overloaded function form a meet semilattice partially ordered by the subtype relation on their domain types, which we call the *more specific than* relation.[3] The Meet Rule guarantees that there are no ambiguous function calls at run time.

We call the first property above the *Return Type Rule* (or *Subtype Rule*). It ensures type preservation when a function call is resolved at run time (based on the ilks of the argument values) to a different (and more specific) definition than the most specific one that could be determined at compile time (based on the types of the argument expressions).

In this paper, we give new Meet and Return Type Rules that ensure safe overloaded functions in a language that supports symmetric multiple dispatch, multiple inheritance, and parametric polymorphism for both types and functions (i.e., generic types and generic functions), as does the Fortress language we are developing [1]. We prove that these rules guarantee type safety. This extends previous work [2] in which we gave analogous rules, and proved the analogous result, for a core of Fortress that does not support generics.

To handle parametric polymorphism, it is helpful to have an intuitive interpretation for generic types and functions. One way to think about a generic 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) generic types, but for many purposes this model of "an infinite set of ground types" is adequate for explanatory purposes. Not so, however, for generic functions.

For some time during the development of Fortress, we considered an interpretation of generic functions analogous to the one above for generic types; that is, the generic function definition:[4]

$$\mathit{tail}[\![X]\!](x\colon \mathrm{List}[\![X]\!])\colon \mathrm{List}[\![X]\!] = e$$

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

$$\mathit{tail}(x\colon \mathrm{List}[\![\mathrm{Object}]\!])\colon \mathrm{List}[\![\mathrm{Object}]\!] = e$$
$$\mathit{tail}(x\colon \mathrm{List}[\![\mathrm{String}]\!])\colon \mathrm{List}[\![\mathrm{String}]\!] = e$$
$$\mathit{tail}(x\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{List}[\![\mathbb{Z}]\!] = e$$
$$\ldots$$

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:

$$\mathit{tail}[\![X]\!](x\colon \mathrm{List}[\![X]\!])\colon \mathrm{List}[\![X]\!] = e_1$$
$$\mathit{tail}(x\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{List}[\![\mathbb{Z}]\!] = e_3$$

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

$$\mathit{tail}(x\colon \mathrm{List}[\![\mathrm{Object}]\!])\colon \mathrm{List}[\![\mathrm{Object}]\!] = e_1$$
$$\mathit{tail}(x\colon \mathrm{List}[\![\mathrm{String}]\!])\colon \mathrm{List}[\![\mathrm{String}]\!] = e_1$$
$$\mathit{tail}(x\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{List}[\![\mathbb{Z}]\!] = e_1$$
$$\ldots$$
$$\mathit{tail}(x\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{List}[\![\mathbb{Z}]\!] = e_3$$

which is ambiguous for calls in which the argument is of type $\mathrm{List}[\![\mathbb{Z}]\!]$.

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

$$\mathit{tail}[\![X]\!](x\colon \mathrm{List}[\![X]\!])\colon \mathrm{List}[\![X]\!] = e_1$$
$$\mathit{tail}[\![X <\colon \mathrm{Number}]\!](x\colon \mathrm{List}[\![X]\!])\colon \mathrm{List}[\![X]\!] = e_2$$

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

It does not suffice to "break ties" by choosing the instantiation of the more specific generic definition. Consider the following overloaded definitions:

$$\mathit{quux}[\![X]\!](x\colon X)\colon \mathbb{Z} = 1$$
$$\mathit{quux}(x\colon \mathbb{Z})\colon \mathbb{Z} = 2$$

Intuitively, we might expect that the call $\mathit{quux}(x)$ evaluates to 2 whenever the ilk of x is a subtype of $\mathbb{Z}$, and to 1

[1] The "domain type" of a function definition is the type of its parameter. Hereafter we consider every function to have a single parameter; the appearance of multiple parameters denotes a single tuple parameter.

[2] A type is a nontrivial subtype of another type if it is not the trivial "bottom" type defined in the next section.

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

[4] The first pair of white square brackets delimits the type parameter declarations, but the other pairs of white brackets provide the type arguments to the generic type List.

otherwise. However, under the "infinite set of monomorphic definitions" interpretation, the call $quux(x)$ when x has type $\mathbb{N} <: \mathbb{Z}$ would evaluate to 1 because the most specific monomorphic definition would be the the instantiation of the generic definition with $\mathbb{N}$.

It is not even always obvious which function definition is the most specific one applicable to a particular call in the presence of overloaded generic functions: 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:

$$foo[\![X <: \mathrm{Object}]\!](x\colon X, y\colon \mathrm{Object})\colon \mathbb{Z} = 1$$
$$foo[\![Y <: \mathrm{Number}]\!](x\colon \mathrm{Number}, y\colon Y)\colon \mathbb{Z} = 2$$

The type parameter of the first definition denotes the type of the first argument, and the type parameter of the second definition denotes the type of the second argument; they bear no relation to each other. 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?

Under the "infinite set of monomorphic definitions" interpretation, these definitions would be equivalent to:

$$foo(x\colon \mathrm{Object}, y\colon \mathrm{Object})\colon \mathbb{Z} = 1$$
$$foo(x\colon \mathrm{Number}, y\colon \mathrm{Object})\colon \mathbb{Z} = 1$$
$$foo(x\colon \mathbb{Z}, y\colon \mathrm{Object})\colon \mathbb{Z} = 1$$
$$\ldots$$
$$foo(x\colon \mathrm{Number}, y\colon \mathrm{Number})\colon \mathbb{Z} = 2$$
$$foo(x\colon \mathrm{Number}, y\colon \mathbb{Z})\colon \mathbb{Z} = 2$$
$$\ldots$$

When foo is called on two arguments of type $\mathbb{Z}$, both $foo(x\colon \mathbb{Z}, y\colon \mathrm{Object})$ and $foo(x\colon \mathrm{Number}, y\colon \mathbb{Z})$ are applicable (assuming $\mathbb{Z} <: \mathrm{Number} <: \mathrm{Object}$). Neither is more specific than the other, and moreover no definition of $foo(x\colon \mathbb{Z}, y\colon \mathbb{Z})$ has been supplied to satisfy the Meet Rule, so this overloading is ambiguous.

We propose to avoid such ambiguities by adopting an alternate model for generic functions, similar to one proposed by Bourdoncle and Merz [3], in which each function definition is regarded not as an infinite set of definitions, but rather as a single definition whose domain type is existentially quantified over its type parameters. (A monomorphic definition is then regarded as a degenerate generic definition.) In this model, overloaded function definitions are (partially) ordered by the subtype relation on existential types. Adapting dispatch and the Meet Rule to use this new partial order is straightforward. Adapting the Return Type Rule is somewhat more complicated, but checking it reduces to checking subtyping relationships between universal types. Adopting this model has made overloaded generic functions in Fortress both tractable and effective. In particular, the overloading of foo just shown is permitted and is not ambiguous, because under this interpretation the second definition is more specific than the first.

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. Unfortunately, this goal is in tension with another requirement, to support modularity and extensibility. In particular, we assume the program will be composed of several modules, and that types defined in one module may be extended by types defined in other modules. We want to be able to check the rules separately for each module, and not have to recheck a module when some other module extends its types.

The difficulty is due to multiple inheritance: Because the type hierarchy defined by a module may be extended by types in other modules, two types may have a common nontrivial subtype even if no type declared in this module extends them both. Thus, for any pair of overloaded function definitions with incomparable domain 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.

Consider, for example, the following overloaded function definitions:

$$print(s\colon \mathrm{String})\colon ()$$
$$print(i\colon \mathbb{Z})\colon ()$$

Although this overloading may seem intuitively to be valid, in a multiple-inheritance type system that allows any type to be extended by some other module, one could define a type $\mathrm{StringAndInteger}$ that extends both String and $\mathbb{Z}$. In that case, a call to $print$ with an argument of type $\mathrm{StringAndInteger}$ would be ambiguous. Thus, this overloading must be rejected by our overloading rules.

To address this problem, Fortress enables programmers to declare "nominal exclusion", restricting how type constructors may be extended, and uses this to derive an *exclusion relation* on types. Types related by exclusion must not have any nontrivial subtype in common. Many languages enforce and exploit exclusion implicitly. For example, single inheritance ensures that incomparable types exclude each other. If the domains of two overloaded function definitions exclude each other, then these definitions can never both be applicable to the same call, so no ambiguity can arise between them. In the example above, if String and $\mathbb{Z}$ exclude each other,[5] then the overloaded definitions of $print$ above are valid.

We already exploited exclusion in our prior work on Fortress without generics, but the constructs Fortress provides for explicitly declaring exclusion are insufficient for

[5] Indeed, String and $\mathbb{Z}$ *are* declared to exclude each other in the Fortress standard library.

allowing some intuitively appealing overloaded functions involving generic types. In particular, we could not guarantee type safety when a type extends multiple instantiations of a generic type. Implicitly forbidding such extension—a property we call *multiple instantiation exclusion*—allows these intuitively appealing overloaded functions.

2. Preliminaries

In this section, we describe the standard parts of the type system we consider in this paper, and establish terminology and notation for entities in this system. To minimize the syntactic overhead, we avoid introducing a new language and instead give a straightforward formalization of the type system. Novel parts of the type system, including the rules for type checking overloaded function declarations, are described in later sections.

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 five forms: *type variables* (ranged over by metavariables X, Y, and Z); *constructed types* (ranged over by metavariables K, L, M and N), consisting of the special constructed type Any and type constructor applications, written $C[\![\overline{T}]\!]$, where C is a type constructor and $\overline{T}$ is a list of types; *structural types*, consisting of arrow types and tuple types; *compound types*, consisting of intersection types and union types; and the special type Bottom, which represents the uninhabited type (i.e., no value belongs to Bottom). The abstract syntax of types is defined as follows (where $\overline{A}$ indicates a possibly empty comma-separated sequence of syntactic elements A):

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

A type may have multiple syntactic forms.[6] In particular, a tuple type of length one is synonymous with its element type, and a tuple type with any Bottom element is synonymous with Bottom. In addition, any types that are provably equivalent as defined below are also synonymous.

As in Fortress, compound types—intersection and union types—and Bottom are *not* first-class: these forms of types 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.

[6] We abuse terminology by not distinguishing type terms and types.

Type checking is done in the context of a *class table* $\mathcal{T}$, which is a set of type constructor declarations (at most one declaration for each type constructor) of the following form:

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

where the only type variables that appear in $\overline{\overline{M}}$ and $\overline{N}$ are those in $\overline{X}$. This declares the type constructor C, and each X_i is a *type parameter* of C with *bounds* $\overline{M_i}$. As usual for languages with nominal subtyping, we allow recursive and mutually recursive references in $\mathcal{T}$ (i.e., a type constructor can be mentioned in the bounds and supertypes of its own and other type constructors' declarations). We say that C *extends* a type constructor D if $N_i = D[\![\overline{T}]\!]$ for some N_i and $\overline{T}$. A class table is *well-formed* if every type that appears in it is well-formed, as defined below, and the extends relation over type constructors is acyclic.

The type constructor declaration above specifies that the constructed type $C[\![\overline{U}]\!]$ (*i*) is *well-formed* (with respect to $\mathcal{T}$) if and only if $|\overline{U}| = |\overline{X}|$ and $U_i <: [\overline{U}/\overline{X}]M_{ij}$ for $1 \le i \le |\overline{U}|$ and $1 \le j \le |\overline{M_i}|$ (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 $1 \le l \le |\overline{N}|$. The 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 Bottom.

We say that a type T (of any form) is *well-formed* with respect to $\mathcal{T}$, and write $T \in \mathcal{T}$, if every constructed type occurring in T is well-formed with respect to $\mathcal{T}$. Typically, the class table is fixed and implicit, and we assume it is well-formed and often omit explicit reference to it.

Given the type constructor declaration above, we denote the set of explicitly declared supertypes of the constructed type $C[\![\overline{T}]\!]$ 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. We also elide the braces delimiting a singleton list of either bounds of a type parameter or supertypes of a class in a type constructor declaration.

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 return types. 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. An intersection type is the most general type that is a subtype of

each of its element types, and a union type is the most specific type that is a supertype of each of its element types.

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 empty, we write this judgment simply as $S <: T$. And we say that the types S and T are *equivalent*, written $S \equiv T$, when $S <: T$ and $T <: S$.

Henceforth, given a type environment, we consider only types whose (free) type variables are bound in the type environment. Because our type language does not involve any type variable binding—type variables are bound only by generic type constructor or function declarations—the set of free type variables of T, written $FV(T)$, is defined as the set of all type variables syntactically occurring within T.

2.2 Extensibility

To enable modular type checking and compilation, 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 from the lack of any such type in the class table. However, we do assume that each declaration is complete, so that all the supertypes of a constructed type are known.

A class table $\mathcal{T}'$ is an *extension* of $\mathcal{T}$ (written $\mathcal{T}' \supseteq \mathcal{T}$) if every declaration in $\mathcal{T}$ is also in $\mathcal{T}'$. From this, it follows that for any well-formed extension $\mathcal{T}'$ of a well-formed class table $\mathcal{T}$, any type that is well-formed with respect to $\mathcal{T}$ is well-formed with respect to $\mathcal{T}'$ and the subtyping relation on $\mathcal{T}'$ agrees with that of $\mathcal{T}$. That is, $T \in \mathcal{T}$ implies $T \in \mathcal{T}'$, and $T <: U$ in $\mathcal{T}$ implies $T <: U$ in $\mathcal{T}'$.

2.3 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. We say that two types T and U have *the same extent* if every value v belongs to T if and only if v belongs to U. No value belongs to Bottom.

We place a requirement on values and on the type system that describes them: Although a value may belong to more than one type, every value v belongs to a unique type $ilk(v)$ (the *ilk* of the value) that is *representable in the type system*[7] and has the property that for every type T, if v belongs to T then $ilk(v) <: T$. (This notion of ilk corresponds to what is sometimes called the "class" or "run-time type" of the value.[8])

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 safe 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 must belong to type U.

2.4 Generic Function Declarations

A function declaration (for a class table) 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 codomain of the function (i.e., the return type). A type parameter declaration consists of a type parameter name and its bounds.

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 the tuple type $\big(\mathrm{List}[\![X]\!], \mathrm{Tree}[\![Y]\!]\big)$, and the return type is $\mathrm{Map}[\![X, Y]\!]$. We abbreviate a function declaration as $f[\![\Delta]\!]\, S : T$ when we do not want to emphasize the bounds. To reduce clutter, we omit the white square brackets of a declaration when the sequence of type parameter declarations is empty, and elide braces around singleton lists of bounds.

A function declaration $d = f[\![\overline{X <: \{\overline{N}\}}]\!]\, S : T$ 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 d with $\overline{W}$. When we do not care about $\overline{W}$, we just say that $f\, U : V$ is an *instance* of d (and it is understood that $U = [\overline{W}/\overline{X}]S$ and $V = [\overline{W}/\overline{X}]T$ for some $\overline{W}$). We use the metavariable $\mathcal{D}$ to range over finite collections of sets of function declarations and $\mathcal{D}_f$ for the subset of $\mathcal{D}$ that contains all declarations of name f.

An instance $f\, U : V$ of a declaration d 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 type T, the set $\mathcal{D}_f(T)$ contains precisely those declarations in $\mathcal{D}_f$ that are applicable to T.

[7] The type system presented here satisfies this requirement simply by providing intersection types.

[8] 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."

3. Overloading Rules and Resolution

In this section, we define the "meaning" of overloaded generic functions; that is, we define how a call to such a function is dispatched, and we give rules for overloaded declarations that ensure that our dispatch procedure is well-defined, as Castagna *et al.* do for overloaded monomorphic functions [4]. The basic idea is simple: For any set of overloaded function declarations, we define a partial order on the declarations—we call this order the *specificity relation*—and dispatch any call to the most specific declaration applicable to the call, based on the ilks of the arguments. The rules for valid overloading ensure that the most specific declaration is well-defined (i.e., unique) for any call (assuming that some declaration is applicable to the arguments), and that the return type of a declaration is a subtype of the return type of any less specific declaration. The latter property is necessary for type preservation for dynamic dispatch: a more specific declaration may be applicable to the ilks of the arguments than the most specific declaration applicable to the static types of the argument expressions, so we must ensure that the return type of this more specific declaration is a subtype of the return type used to type check the program (i.e., at compile time).

Specifically, we define three rules:[9] the *No Duplicates Rule* ensures that no two declarations are equally specific; the *Meet Rule* ensures that the set of overloaded declarations form a meet semilattice under the specificity relation; and the *Return Type Rule* ensures type preservation for dynamic dispatch. We prove that any set of overloaded function declarations satisfying these three properties is safe, even if the class table is extended (Theorem 1).

3.1 Specificity of Generic Function Declarations

For monomorphic function declarations, the specificity relation is just subtyping on their domain types. However, the domain type of a generic function declaration may include type parameters of the declaration, and type parameters of distinct declarations bear no particular relation to each other. Furthermore, the subtyping relation between their domain types of may depend on the instantiation of their type parameters, as illustrated by foo and $quux$ in the introduction.

Instead of using subtyping, we adopt the following intuitive notion of specificity: One declaration is more specific than another if the second is applicable to every argument that the first is applicable to. That is, for any $d_1, d_2 \in \mathcal{D}_f$, d_1 is *more specific* than d_2 (written $d_1 \preceq d_2$) if $d_1 \in \mathcal{D}_f(T)$ implies $d_2 \in \mathcal{D}_f(T)$ for every well-formed type T. Neatly, this turns out to be equivalent subtyping over domain types where the domain type of a generic function declaration is interpreted as an existential type [3]; we use that formulation to mechanically check the overloading rules (see Section 6).

[9] The meet rule of Castagna *et al.* requires the existence and uniqueness of the meet. We split these into two rules.

This definition of specificity introduces a type inference problem for dynamic dispatch: If d_2 is the most specific declaration applicable to the static types of the argument expressions, and $d_1 \preceq d_2$ is the most specific declaration applicable to the ilks of the arguments, then the type parameter instantiations derived by static type inference are relevant to d_2, but not to d_1. Because the call is dispatched to d_1, we require type parameters for d_1 to be inferred *dynamically*. Showing how to do so is beyond the scope of this paper.

3.2 Overloading Rules

Given a class table $\mathcal{T}$, a set $\mathcal{D}$ of generic function declarations for $\mathcal{T}$, and a function name f, the set $\mathcal{D}_f$ is *valid* (or is a *valid overloading*) if it satisfies the following three rules:

No Duplicates Rule For every $d_1, d_2 \in \mathcal{D}_f$, if $d_1 \preceq d_2$ and $d_2 \preceq d_1$ then $d_1 = d_2$.

Meet Rule For every $d_1, d_2 \in \mathcal{D}_f$, there exists a declaration $d_0 \in \mathcal{D}_f$ (possibly d_1 or d_2) such that, $d_0 \preceq d_1$ and $d_0 \preceq d_2$ and d_0 is applicable to any type $T \in \mathcal{T}$ to which both d_1 and d_2 are applicable.

Return Type Rule For every $d_1, d_2 \in \mathcal{D}_f$ with $d_1 \preceq d_2$, and every type $T \not\equiv \mathrm{Bottom}$ such that $d_1 \in \mathcal{D}_f(T)$, if an instance $f\, S_2 : T_2$ of d_2 is applicable to T, then there is an instance $f\, S_1 : T_1$ of d_1 that is applicable to T with $T_1 <: T_2$.

The No Duplicates Rule forbids distinct declarations from being equally specific (i.e., each more specific than the other).

The Meet Rule requires every pair of declarations to have a *disambiguating declaration*, which is more specific than both and applicable whenever both are applicable. (If one of the pair is more specific than the other, then it is the disambiguating declaration.)

The Return Type Rules guarantees that whenever the type checker might have used an instance of a declaration d_2 to check a program, and then a more specific declaration d_1 is selected by dynamic dispatch, then there is some instance of d_1 that is applicable to the argument and whose return type is a subtype of the return type of the instance of d_2 the type checker used, which is necessary for type preservation, as discussed above.

Since Bottom is well-formed, and tuple types with different numbers of arguments have no common subtype other than Bottom, the Meet Rule requires that an overloaded function with declarations that take different numbers of arguments have a declaration applicable only to Bottom. Such a declaration would never be applied (because no value belongs to Bottom), and it cannot be written in Fortress (because Bottom is not first-class). To avoid this technicality, we implicitly augment every set $\mathcal{D}_f$ with a declaration $f\,\mathrm{Bottom} : \mathrm{Bottom}$. This declaration is strictly more specific than any declaration that a programmer can write, and its re-

turn type is a subtype of every type, so it trivially satisfies all three rules when checked with any other declaration in $\mathcal{D}_f$.

This technicality raises the following question: Must the Meet Rule hold for every $T \in \mathcal{T}$? Could we not, for example, have excluded Bottom from consideration, as in the Return Type Rule, and avoided the technicality? If so, for which types is it necessary that the Meet Rule hold? The answer is, we must check the Meet Rule for a type $T \in \mathcal{T}$ to which both d_1 and d_2 are applicable if there could be a value of type T such that for any type $T' \in \mathcal{T}$ to which the value belongs, $T <: T'$. In other words, we must check it for all "leaf" types. Thus, if we did not require extensibility, we can check the Meet Rule only for those types that are ilks of values. However, because we require extensibility, and we support multiple inheritance, we use intersection types instead.

3.3 Properties of Overloaded Functions

With the rules for valid overloading laid out, we now describe some useful properties of valid overloaded sets and of the rules themselves.

Lemma 1 If d_1 and d_2 are declarations in $\mathcal{D}_f$ such that $d_1 \preceq d_2$ and $d_2 \not\preceq d_1$, then the pair (d_1, d_2) satisfies the No Duplicates Rule and the Meet Rule.

Proof: The No Duplicates Rule is vacuously satisfied, and the Meet Rule is satisfied with $d_0 = d_1$ since $d_1 \preceq d_2$ implies that d_1 is applicable to a type T if and only if both d_1 and d_2 are applicable to T. □

Lemma 2 For every type $T \in \mathcal{T}$, if $\mathcal{D}_f$ is a valid set with respect to $\mathcal{T}$ then so is $\mathcal{D}_f(T)$.

Proof: The No Duplicates Rule and Return Type Rule are straightforward applications of the respective rules on $\mathcal{D}_f$.

Let d_1, d_2 be declarations in $\mathcal{D}_f(T)$ and let $d_0 \in \mathcal{D}_f$ be its disambiguating declaration guaranteed by the Meet Rule on $\mathcal{D}_f$. Then d_0 is applicable to exactly those types U to which d_1 and d_2 are both applicable. Since d_1 and d_2 are by definition both applicable to T, d_0 must also be applicable to T, and hence $d_0 \in \mathcal{D}_f(T)$. Therefore the Meet Rule on $\mathcal{D}_f(T)$ is satisfied. □

To further characterize valid sets of overloaded definitions and the more specific relation $\preceq$, we interpret them as meet semilattices. A partially ordered set $(A, \sqsubseteq)$ forms a *meet semilattice* if, for every pair of elements $a, b \in A$, their greatest lower bound, or *meet*, is also in A.

Lemma 3 A valid set of overloaded function declarations forms a meet semilattice with the more specific relation.

Proof: Suppose $\mathcal{D}_f$ is a valid set of overloaded function declarations with respect to class table $\mathcal{T}$. First, $(\mathcal{D}_f, \preceq)$ forms a partially ordered set: clearly $\preceq$ is reflexive and transitive, and antisymmetry is a direct corollary of the No Duplicates Rule.

Second, we must show that (i) for every $d_1, d_2 \in \mathcal{D}_f$ there exists a $d_0 \in \mathcal{D}_f$ such that $d_0 \preceq d_1$ and $d_0 \preceq d_2$ and (ii) if there exists a $d_0' \in \mathcal{D}_f$ such that $d_0' \preceq d_1$ and $d_0' \preceq d_2$ then $d_0' \preceq d_0$.

Let d_1 and d_2 be declarations in $\mathcal{D}_f$. By the Meet Rule, there exists a declaration $d_0 \in \mathcal{D}_f$ that is applicable to a type $T \in \mathcal{T}$ if and only if both d_1 and d_2 are too. Since for every T to which d_0 is applicable we have that d_1 and d_2 are also applicable to it, we know that $d_0 \preceq d_1$ and $d_0 \preceq d_2$.

Now let $d_0' \in \mathcal{D}_f$ be more specific than both d_1 and d_2. Then for every type $T \in \mathcal{T}$ such that d_0' is applicable to T, d_1 and d_2 are also applicable to T; thus d_0 is applicable to T and $d_0' \preceq d_0$. □

The No Duplicates Rule and the Meet Rule each corresponds to a defining property of meet semilattices (antisymmetry and the existence of meets, respectively), while the Return Type Rule guarantees that this interpretation is consistent with the semantics of multiple dynamic dispatch.

Lemma 4 A valid set of overloaded function declarations $\mathcal{D}_f(T)$ has a unique most specific declaration.

Proof sketch: The set $\mathcal{D}_f(T)$ forms a meet semilattice by the previous lemma and moreover it is clearly finite. By straightforward induction a finite meet semilattice has a least element, so there exists a unique declaration in $\mathcal{D}_f(T)$ that is more specific than all others. □

3.4 Overloading Resolution Safety

We now prove the main theorem of this paper, that a valid set of overloaded generic function declarations is safe even if the class table is extended. Before proving the theorem we establish two lemmas. First, we show that if a set of declarations is valid for a given class table, then it is valid for any (well-formed) extension of that class table. Second, we show that if a set of overloaded declarations is valid, then there is always a single best choice of declaration to which to dispatch any (legal) call to that function (i.e., the unique most specific declaration applicable to the arguments).

Lemma 5 (Extensibility) If $\mathcal{D}_f$ is valid with respect to the class table $\mathcal{T}$, then $\mathcal{D}_f$ is valid with respect to any extension $\mathcal{T}'$ of $\mathcal{T}$.

Proof sketch: In Section 6 we will show that checking the validity of $\mathcal{D}_f$ can be reduced to examining subtype relationships between existential and universal types, which are constructed solely from types appearing in $\mathcal{D}_f$ and hence $\mathcal{T}$. Extension of the class table preserves subtype relationships between types in $\mathcal{T}$ and hence preserves validity of $\mathcal{D}_f$. □

Lemma 6 (Unambiguity) If $\mathcal{D}_f$ is valid with respect to the class table $\mathcal{T}$, then for every type $T \in \mathcal{T}$ such that $\mathcal{D}_f(T)$

is nonempty, there is a unique most specific declaration in $\mathcal{D}_f(T)$.

Proof: Let $T \in \mathcal{T}$ be a type such that $\mathcal{D}_f(T)$ is nonempty. This set is valid by Lemma 2 and thus contains a unique most specific declaration by Lemma 4. □

Theorem 1 (Overloading Safety) Suppose $\mathcal{D}_f$ is valid with respect to the class table $\mathcal{T}$. Then for any type $S \in \mathcal{T}' \supseteq \mathcal{T}$, if $\mathcal{D}_f(S)$ is nonempty then there exists a unique most specific declaration $d_S \in \mathcal{D}_f(S)$. Furthermore for any declaration $d \in \mathcal{D}_f(S)$ and instance $f\,T\!:\!U$ of d, there exists an instance $f\,V\!:\!W$ of d_S that is applicable to S such that $W <: U$.

Proof: The Extensibility Lemma lets us consider only the case when $\mathcal{T}' = \mathcal{T}$. Now the Unambiguity Lemma entails that such a d_S exists, and the Return Type Rule entails the rest. □

4. Exclusion

Although the rules in Section 3 allow programmers to write valid sets of overloaded generic function declarations, they sometimes reject overloaded definitions that might seem to be valid. For example, given the type system as we have described it thus far, the overloaded $tail$ function from the introduction would be rejected by the overloading rules.

These are not false negatives: multiple inheritance can introduce ambiguities by extending two incomparable types, as discussed in the introduction. Because we allow class tables to be extended by unknown modules, we cannot generally infer that two types have no common nontrivial subtype from the lack of any such declared type. Therefore, the Meet Rule requires the programmer to provide a disambiguating definition for any pair of overloaded definitions whose domain types are incomparable.

This problem is not new with parametric polymorphism, as the $print$ example in the introduction shows. To address the problem, Fortress defines an *exclusion relation* $\Diamond$ over types such that two types that exclude each other have no common nontrivial subtypes; that is, if $T \Diamond U$ then $T \cap U$ is synonymous with Bottom. Thus, overloaded definitions whose domain types exclude each other trivially satisfy the Meet Rule: there are no types (other than Bottom) to which both definitions are applicable. Exclusion allows us to describe explicitly what is typically implicit in single-inheritance class hierarchies.

In our previous work on Fortress without generics [2], we provided a special rule—the *Exclusion Rule*—to exploit this information. However, the Exclusion Rule can also be viewed, as we do in this paper, as a special case of the Meet Rule, where there are no nontrivial types to which both definitions are applicable.

Fortress provides three mechanisms to explicitly declare exclusion [1]: an `object` declaration, a `comprises` clause and an `excludes` clause. We describe these precisely in Section 4.1, but for now, we simply note that they do not help with the overloaded $tail$ function. Specifically, even with these exclusion mechanisms, we cannot define List so that the definitions for $tail$ satisfy the Return Type Rule.

To see this, consider the following overloaded definitions (from Section 1):

$$tail[\![T]\!](x\colon \mathrm{List}[\![T]\!])\colon \mathrm{List}[\![T]\!]$$
$$tail(x\colon \mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{List}[\![\mathbb{Z}]\!]$$

and the following type constructor declaration:

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

Both definitions of $tail$ are applicable to the type $\mathrm{BadList}$, and the monomorphic one is more specific. Two instances of the generic definition are applicable to this type:

$$tail(\mathrm{List}[\![\mathbb{Z}]\!])\colon \mathrm{List}[\![\mathbb{Z}]\!]$$
$$tail(\mathrm{List}[\![\mathrm{String}]\!])\colon \mathrm{List}[\![\mathrm{String}]\!]$$

The Return Type Rule requires that the return type of each of these instances be a supertype of the return type of the monomorphic definition (the monomorphic definition is its only one instance); that is, it requires $\mathrm{List}[\![\mathbb{Z}]\!] <: \mathrm{List}[\![\mathbb{Z}]\!]$ and $\mathrm{List}[\![\mathbb{Z}]\!] <: \mathrm{List}[\![\mathrm{String}]\!]$. The latter is clearly false.

A similar issue arises when trying to satisfy the Meet Rule by providing a disambiguating definition for incomparable definitions, as in the following example (where $\mathbb{Z} <: \mathbb{R}$):

$$minimum[\![X <: \mathbb{R}, Y <: \mathbb{Z}]\!](p\colon \mathrm{Pair}[\![X, Y]\!])\colon \mathbb{R}$$
$$minimum[\![X <: \mathbb{Z}, Y <: \mathbb{R}]\!](p\colon \mathrm{Pair}[\![X, Y]\!])\colon \mathbb{R}$$
$$minimum[\![X <: \mathbb{Z}, Y <: \mathbb{Z}]\!](p\colon \mathrm{Pair}[\![X, Y]\!])\colon \mathbb{Z}$$

The first definition is applicable to exactly those arguments of type $\mathrm{Pair}[\![X, Y]\!]$ for some $X <: \mathbb{R}$ and $Y <: \mathbb{Z}$; the second is applicable to exactly those arguments of type $\mathrm{Pair}[\![X, Y]\!]$ for some $X <: \mathbb{Z}$ and $Y <: \mathbb{R}$. So we might think that both definitions are applicable to exactly those arguments of type $\mathrm{Pair}[\![X, Y]\!]$ for some $X <: \mathbb{Z}$ and $Y <: \mathbb{Z}$, which is exactly when the third definition is applicable. However, this is not true! We could, for example, have the following type constructor declaration:

$$\mathrm{BadPair} <: \{\, \mathrm{Pair}[\![\mathbb{R}, \mathbb{Z}]\!], \mathrm{Pair}[\![\mathbb{Z}, \mathbb{R}]\!] \,\}$$

The first two definitions of $minimum$ are both applicable to $\mathrm{BadPair}$ but the third is not.

We might say that the problem with the above examples is not with the definitions of $tail$ and $minimum$, but with the definitions of $\mathrm{BadList}$ and $\mathrm{BadPair}$; we must reject the idea that a value may belong to $\mathrm{List}[\![\mathbb{Z}]\!]$ or $\mathrm{List}[\![\mathrm{String}]\!]$ — or to $\mathrm{Pair}[\![\mathbb{Z}, \mathbb{R}]\!]$ or $\mathrm{Pair}[\![\mathbb{R}, \mathbb{Z}]\!]$ —but not to both. Indeed, Fortress imposes a rule that forbids *multiple instantiation inheritance* [8], in which a type (other than Bottom) is a subtype of distinct applications of a type constructor.[10] We

[10] This definition suffices for the type system described in this paper, in which all type parameters are invariant. It is straightforward, but beyond the scope of this paper, to extend this definition to systems that support covariant and contravariant type parameters.

call this rule *multiple instantiation exclusion* and adopt it here.

Multiple instantiation exclusion is easy to enforce statically, and experience suggests that it is not onerous in practice: it is already required in Java, for example [7]. Also, Kennedy and Pierce have shown that in systems that enforce multiple instantiation exclusion (along with some technical restrictions), nominal subtyping is decidable [8].[11]

4.1 Well-Formed Class Tables with Exclusion

To incorporate exclusion into our type system, we first augment type constructor declarations with two (optional) clauses—the `excludes` and `comprises` clauses—and add a new kind of type constructor declaration—the `object` declaration. We then change the definition of well-formed class tables to reflect the new features, and to enforce multiple instantiation exclusion.

The syntax for a type constructor declaration with the optional `excludes` and `comprises` clauses, each of which specifies a list of types, is:

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

This declaration asserts that for well-formed type $C[\![\overline{T}]\!]$, the only common subtype of $C[\![\overline{T}]\!]$ and $[\overline{T}/\overline{X}]L_i$ for any L_i in $\overline{L}$ is Bottom, and any strict subtype of $C[\![\overline{T}]\!]$ must also be a subtype of $[\overline{T}/\overline{X}]K_i$ for some K_i in $\overline{K}$.

Omitting the `excludes` clause is equivalent to having `excludes` $\{\}$; omitting the `comprises` clause is equivalent to having `comprises` $\{\,\mathrm{Any}\,\}$.[12]

We define the sets of instantiations of types in `excludes` and `comprises` clauses analogously to $C[\![\overline{T}]\!].\mathit{extends}$. That is, for an application $C[\![\overline{T}]\!]$ of the declaration above, we have:

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

A class table may also include `object` declarations, which have the following syntax:

$$\texttt{object}\ D[\![\overline{X <: \{\overline{M}\}}]\!] <: \{\overline{N}\}$$

This declaration is convenient for defining "leaf types": it asserts that $D[\![\overline{T}]\!]$ has no subtypes other than itself and Bottom. Although the declaration has no `excludes` or `comprises` clause, this condition implies that $D[\![\overline{T}]\!]$ excludes any type other than its supertypes (and therefore it is as if it had a clause `comprises` $\{\}$).

[11] They also show that forbidding contravariant type parameters results in decidable nominal subtyping, so subtyping in our type system is decidable in any case.

[12] To catch likely programming errors, Fortress 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.

Multiple instantiation exclusion further restricts generic types: Distinct instantiations of a generic type (i.e., distinct applications of a type constructor) have no common subtype other than Bottom.

To define well-formedness for class tables with exclusion (including multiple instantiation exclusion), we define an *exclusion relation* $\lozenge$ over well-formed types: $S \lozenge T$ asserts that S and T have no common subtypes other than Bottom. For constructed types $C[\![\overline{T}]\!]$ and $D[\![\overline{U}]\!]$, $C[\![\overline{T}]\!] \lozenge D[\![\overline{U}]\!]$ if

- $D[\![\overline{U}]\!] \in C[\![\overline{T}]\!].\mathit{excludes}$;
- for all $L \in C[\![\overline{T}]\!].\mathit{comprises}$, $D[\![\overline{U}]\!]$ is not a subtype of L;
- $C[\![\overline{T}]\!]$ is declared by an `object` declaration and $C[\![\overline{T}]\!]$ is not a subtype of $D[\![\overline{U}]\!]$;
- $D[\![\overline{U}]\!] \lozenge C[\![\overline{T}]\!]$ by any of the conditions above;
- $C = D$ and $\overline{T} \not\equiv \overline{U}$; or
- $M \lozenge N$ for some $M :> C[\![\overline{T}]\!]$ and $N :> D[\![\overline{U}]\!]$.

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 well-formed constructed types M and N, if $M \lozenge N$ then no well-formed constructed type is a subtype of both M and N. A well-formed extension to a class table $\mathcal{T}$ must preserve this property.

Except for the imposition of multiple instantation exclusion, these changes generalize the standard type system described in Section 2: A class table that does not use any of the new features is well-formed in this augmented system exactly when it is well-formed in the standard system. On the other hand, multiple instantiation exclusion restricts the set of well-formed class tables: a table that is well-formed when multiple instantiation inheritance is permitted might not be well-formed under multiple instantiation exclusion.

We extend the exclusion relation to structural and compound types as follows: Every arrow type excludes every non-arrow type other than Any. Every non-singleton tuple type excludes every non-tuple type other than Any. (A singleton tuple type is synonymous with its element type, and so excludes exactly those types excluded by its element type.) Non-singleton tuple type $(\overline{V})$ excludes non-singleton tuple type $(\overline{W})$ 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; a union type excludes any type excluded by *all* of its constituent types. Bottom excludes every type (including itself—it is the only type that excludes itself), and Any does not exclude any type other than Bottom. (We define the exclusion relation formally in Section 7.)

5. Examples

We now consider several sets of overloaded generic function declarations, and argue informally why they are (or are not, in one case) permitted by the rules in Section 3, paying particular attention to where multiple instantiation exclusion

is required. We give a formal system and algorithm for performing these checks in Section 6.

First, consider the function *foo* from Section 1:

$$foo[\![X <: \mathrm{Object}]\!](x\colon X, y\colon \mathrm{Object})\colon \mathbb{Z} = 1$$
$$foo[\![Y <: \mathrm{Number}]\!](x\colon \mathrm{Number}, y\colon Y)\colon \mathbb{Z} = 2$$

This overloading is valid: The second definition is strictly more specific than the first because the first definition is applicable to a pair of arguments exactly if the type of each is a subtype of Object, whereas the second is applicable to a pair of arguments exactly if the type of each is a subtype of Number. Thus, these definitions satisfy the No Duplicates Rule and the Meet Rule by Lemma 1. And they satisfy the Return Type Rule because the return type of both definitions is always $\mathbb{Z}$.

The overloaded definitions for *tail* are also valid:

$$tail[\![X]\!]\big(x\colon \mathrm{List}[\![X]\!]\big)\colon \mathrm{List}[\![X]\!] = e_1$$
$$tail[\![X <: \mathrm{Number}]\!]\big(x\colon \mathrm{List}[\![X]\!]\big)\colon \mathrm{List}[\![X]\!] = e_2$$
$$tail\big(x\colon \mathrm{List}[\![\mathbb{Z}]\!]\big)\colon \mathrm{List}[\![\mathbb{Z}]\!] = e_3$$

The first definition is applicable to any argument of type $\mathrm{List}[\![T]\!]$ for any well-formed type T, the second is applicable to an argument of type $\mathrm{List}[\![T]\!]$ when $T <: \mathrm{Number}$, and the third is applicable to an argument of type $\mathrm{List}[\![\mathbb{Z}]\!]$. Thus, each definition is strictly more specific than the preceding one, so the No Duplicates Rule and Meet Rule are satisfied for each pair of definitions by Lemma 1.

To see that the Return Type Rule is satisfied by the first two definitions, consider any type $W \not\equiv \mathrm{Bottom}$ to which the second definition is applicable—so $W <: \mathrm{List}[\![T]\!]$ for some $T <: \mathrm{Number}$—and any instantiation of the first definition with type U that is applicable to W—so $W <: \mathrm{List}[\![U]\!]$. Then $W <: \mathrm{List}[\![T]\!] \cap \mathrm{List}[\![U]\!]$. By multiple instantiation exclusion, $\mathrm{List}[\![T]\!] \cap \mathrm{List}[\![U]\!] \equiv \mathrm{Bottom}$ unless $T \equiv U$. Since $W \not\equiv \mathrm{Bottom}$, we have $T \equiv U$, so $W <: \mathrm{List}[\![U]\!]$ with $U <: \mathrm{Number}$. Thus, the instantiation of the second definition with U has return type $\mathrm{List}[\![U]\!]$, which is also the return type of the instantiation of the first definition under consideration. (In Section 7.8 we describe how to incorporate this sort of reasoning about validity into our algorithmic checking of the overloading rules.)

The Return Type Rule is also satisfied by the third definition and either of the first two because the third definition is applicable only to arguments of type $\mathrm{List}[\![\mathbb{Z}]\!]$, and because of multiple instantiation exclusion, the only instantiation of either the first or second definition that is applicable to such an argument is its instantiation with $\mathbb{Z}$. That instantiation has return type $\mathrm{List}[\![\mathbb{Z}]\!]$, which is also the return type of the third definition.

The *minimum* example from Section 4 is also valid under multiple instantiation exclusion, which is necessary in this case to satisfy the Meet Rule rather than the Return Type Rule:

$$minimum[\![X <: \mathbb{R}, Y <: \mathbb{Z}]\!]\big(p\colon \mathrm{Pair}[\![X, Y]\!]\big)\colon \mathbb{R}$$
$$minimum[\![X <: \mathbb{Z}, Y <: \mathbb{R}]\!]\big(p\colon \mathrm{Pair}[\![X, Y]\!]\big)\colon \mathbb{R}$$
$$minimum[\![X <: \mathbb{Z}, Y <: \mathbb{Z}]\!]\big(p\colon \mathrm{Pair}[\![X, Y]\!]\big)\colon \mathbb{Z}$$

Any argument to which the first two definitions are both applicable must be of type $\mathrm{Pair}[\![X_1, Y_1]\!] \cap \mathrm{Pair}[\![X_2, Y_2]\!]$ for some $X_1 <: \mathbb{R}$, $Y_1 <: \mathbb{Z}$, $X_2 <: \mathbb{Z}$, and $Y_2 <: \mathbb{R}$. Multiple instantiation exclusion implies that $X_1 = X_2$ and $Y_1 = Y_2$, so the argument must be of type $\mathrm{Pair}[\![X_1, Y_1]\!]$, where $X_1 <: \mathbb{R} \cap \mathbb{Z} = \mathbb{Z}$ and $Y_1 <: \mathbb{Z} \cap \mathbb{R} = \mathbb{Z}$, so the third definition is applicable to it. And since the third definition is more specific than the first two, it satisfies the requirement of the Meet Rule.

The following set of overloaded declarations is also valid (given the declaration $\mathrm{ArrayList}[\![X]\!] <: \mathrm{List}[\![X]\!]$):

$$bar[\![X]\!]\mathrm{ArrayList}[\![X]\!]\colon \mathbb{Z}$$
$$bar[\![Y <: \mathbb{Z}]\!]\mathrm{List}[\![Y]\!]\colon \mathbb{Z}$$
$$bar[\![Z <: \mathbb{Z}]\!]\mathrm{ArrayList}[\![Z]\!]\colon \mathbb{Z}$$

The first two declarations are incomparable: the first is applicable to $\mathrm{ArrayList}[\![T]\!]$ for any type T, the second to $\mathrm{List}[\![U]\!]$ for $U <: \mathbb{Z}$. Thus, both declarations are applicable to any argument of type $\mathrm{ArrayList}[\![T]\!] \cap \mathrm{List}[\![U]\!]$ for any T and $U <: \mathbb{Z}$. Since $\mathrm{ArrayList}[\![T]\!] <: \mathrm{List}[\![T]\!]$, this type is a subtype of $\mathrm{List}[\![T]\!] \cap \mathrm{List}[\![U]\!]$, which, because of multiple instantiation exclusion, is Bottom unless $T \equiv U$, in which case, $\mathrm{ArrayList}[\![T]\!] \cap \mathrm{List}[\![U]\!] \equiv \mathrm{ArrayList}[\![U]\!]$. This is exactly the type to which the third declaration is applicable, so the Meet Rule is satisfied.

Note that this example is similar to the previous one with *minimum* except that rather than having each of the two definitions being more restrictive on a type parameter, one uses a more specific type constructor.

Finally, we consider three examples that do not involve generic types, beginning with the following declarations:

$$baz[\![X]\!](x\colon X)\colon X$$
$$baz(x\colon \mathbb{Z})\colon \mathbb{Z}$$

This pair is *not* valid: it does not satisfy the Return Type Rule. Consider, for example, an argument of type $\mathbb{N} <: \mathbb{Z}$. The second declaration, which is more specific than the first, and the instantiation of the first declaration with $\mathbb{N}$ are both applicable to this argument, but the return type $\mathbb{Z}$ of the second declaration is not a subtype of the return type $\mathbb{N}$ of the instance of the first declaration. This rejection by the Return Type Rule is not gratuitous: *baz* may be called with an argument of type $\mathbb{N}$ in a context that expects an $\mathbb{N}$ in return.

We can fix this example by making the second declaration generic:

$$baz[\![X]\!](x\colon X)\colon X$$
$$baz[\![X <: \mathbb{Z}]\!](x\colon X)\colon X$$

This pair is valid: the second declaration is strictly more specific than the first, so the No Duplicates and Meet Rules are satisfied. To see that the Return Type Rule is satisfied, consider any type $W \not\equiv \mathrm{Bottom}$ to which the second declaration is applicable—so $W <: \mathbb{Z}$—and any instantiation

of the first with some type T that is applicable to W—so $W <: T$. Then the instantiation of the second declaration with W has return type W, which is a subtype of the return type T of the instance of the first declaration under consideration.

6. Overloading Rules Checking

In this section, we describe how to mechanically check the overloading rules from Section 3. The key insight is that the more specific relation on overloaded function declarations corresponds to the subtyping relation on the domain types of the declarations, where the domain types of generic function declarations are *existential types* [3]. Thus, the problem of determining whether one declaration is more specific than another reduces to the problem of determining whether one existential type is a subtype of another.

We then formulate the overloading rules as subtyping checks on existential and universal types (universal types arise in the reformulation of the Return Type Rule), and give an algorithm to perform these subtyping checks. The algorithm we describe is sound but not complete: it does rejects some sets of overloaded functions that are valid by the overloading rules in Section 3, but it accepts many of them, including all of the valid examples in Section 5.

6.1 Existential and Universal Types

Given a generic function declaration $d = f[\![\Delta]\!]\, S{:}T$, its domain type, written $dom(d)$, is the existential type $\exists[\![\Delta]\!]S$. An *existential type* binds type parameter declarations over a type, but these type parameters cannot be instantiated; instead, the existential type represents some hidden type instantiation and the corresponding instantiated type. We write $\exists[\![\overline{X <: \{N\}}]\!]T$ to bind each type variable X_i with bounds $\{\overline{N_i}\}$ over the type T, and we use the metavariable δ to range over existential types.

The arrow type of declaration d above, written $arrow(d)$, is the universal arrow type $\forall[\![\Delta]\!]S \to T$. We use this to formulate the Return Type Rule. A *universal type* binds type parameter declarations over some type and can be instantiated by any types fitting the type parameters' bounds. We write $\forall[\![\overline{X <: \{N\}}]\!]T$ to bind 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.

Note that universal and existential types are *not* actually types in our system.

6.2 Universal and Existential Subtyping

We define subtyping judgments for universal and existential types, which we use in checking the overloading rules. We actually define inner and outer subtyping judgments on universals and existentials; the former correspond to a relatively standard interpretation of each (which resembles those defined in [3]); the latter incorporate *quantifier reduction*, defined in Section 7.8.

Existential Subtyping: $\Delta \vdash \delta \lesssim \delta \quad \Delta \vdash \delta \leq \delta$

$$\frac{\Delta' = \Delta, \overline{X <: \{\overline{M}\}} \qquad \overline{X} \cap (FV(U) \cup FV(\overline{N})) = \emptyset \\ \Delta' \vdash T <: [\overline{V}/\overline{Y}]U \qquad \forall i\,.\, \Delta' \vdash V_i <: [\overline{V}/\overline{Y}]\{\overline{N_i}\}}{\Delta \vdash \exists[\![\overline{X <: \{\overline{M}\}}]\!]T \lesssim \exists[\![\overline{Y <: \{\overline{N}\}}]\!]U}$$

$$\frac{\Delta \vdash \delta \xrightarrow{\equiv} \delta_r \qquad \Delta \vdash \delta_r \lesssim \delta'}{\Delta \vdash \delta \leq \delta'}$$

Universal Subtyping: $\Delta \vdash \sigma \lesssim \sigma \quad \Delta \vdash \sigma \leq \sigma$

$$\frac{\Delta' = \Delta, \overline{Y <: \{\overline{N}\}} \qquad \overline{Y} \cap (FV(T) \cup FV(\overline{M})) = \emptyset \\ \Delta' \vdash [\overline{V}/\overline{X}]T <: U \qquad \forall i\,.\, \Delta' \vdash V_i <: [\overline{V}/\overline{X}]\{\overline{M_i}\}}{\Delta \vdash \forall[\![\overline{X <: \{\overline{M}\}}]\!]T \lesssim \forall[\![\overline{Y <: \{\overline{N}\}}]\!]U}$$

$$\frac{\Delta \vdash \sigma' \xrightarrow{\equiv} {\sigma'}_r \qquad \Delta \vdash \sigma \lesssim {\sigma'}_r}{\Delta \vdash \sigma \leq \sigma'}$$

Figure 1. Subtyping of universal and existential types. Note that alpha-renaming of type variables may be necessary to apply these rules.

The inner subtyping judgment on existentials $\Delta \vdash \delta_1 \lesssim \delta_2$, defined in Figure 1, states that δ_1 is a subtype of δ_2 in the environment Δ if the constituent type of δ_1 is a subtype of an instance of δ_2 in the environment obtained by conjoining Δ and the bounds of δ_1.

In the outer subtyping judgment $\Delta \vdash \delta \leq \delta'$, we first perform *existential reduction* to produce δ_r (denoted $\Delta \vdash \delta \xrightarrow{\equiv} \delta_r$). Then we check whether $\Delta \vdash \delta_r \lesssim \delta'$. We provide (and explain) the formal definition of existential reduction in Section 7.8, but for now note that it has the following properties:

1. $\Delta \vdash \delta_r \lesssim \delta$
2. $\Delta \vdash \delta \lesssim \delta'$ implies $\Delta \vdash \delta_r \lesssim {\delta'}_r$
3. $(\delta_r)_r = \delta_r$
4. $(\exists[\![]\!]T)_r = \exists[\![]\!]T$

The first three properties show that $\leq$ is a preorder and that $\lesssim$ implies $\leq$. Adding the fourth property lets us show that any ground instance T of δ with $T \neq \mathrm{Bottom}$ is an instance of δ_r

We use existential reduction because merely extending the subtyping relation for ordinary types with exclusion is not enough to let us check the overloading rules. For example, to check that the first two declarations of $\mathcal{D}_{bar}$ from Section 5 satisfy the Meet Rule, we must be able to deduce that the existential

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

and the existential

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

describe the same set of ground instances of types.

The rules for universals are dual to those for existentials. The inner subtyping judgment on universals $\Delta \vdash \sigma_1 \lesssim \sigma_2$, defined in Figure 1, states that σ_1 is a subtype of σ_2 in the environment Δ, if an instance of σ_1 is a subtype of the constituent type of σ_2 in the environment obtained by conjoining Δ and the bounds of σ_2. In the outer universal subtyping judgment $\Delta \vdash \sigma \leq \sigma'$, we first perform *universal reduction* to produce ${\sigma'}_r$ (denoted $\Delta \vdash \sigma' \xrightarrow{\equiv} {\sigma'}_r$) and then check whether $\Delta \vdash \sigma \lesssim {\sigma'}_r$. Again, we provide the formal definition of universal reduction in Section 7.8, noting that it has the following properties:

1. $\Delta \vdash \sigma \lesssim \sigma_r$
2. $\Delta \vdash \sigma \lesssim \sigma'$ implies $\Delta \vdash \sigma_r \lesssim {\sigma'}_r$
3. $(\sigma_r)_r = \sigma_r$
4. $(\forall[\![]\!] S \to T)_r = \forall[\![]\!] S \to T$

Again the first three properties show that $\leq$ is a preorder and that $\lesssim$ implies $\leq$. Adding the fourth property lets us show that any ground instance $S \to T$ of δ with $S \neq \mathrm{Bottom}$ is an instance of σ_r.

We need universal reduction for the same reason we need existential reduction, to check the overloading rules. For example to show the first two declarations in $\mathcal{D}_{tail}$ from Section 5 satisfy the Return Type Rule, we use universal reduction to show that

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

and

$$\forall[\![W <: \mathrm{Number}]\!]\mathrm{List}[\![W]\!] \to \mathrm{List}[\![W]\!]$$

have all the same nontrivial instances.

6.3 Mechanically Checking the Rules

With our interpretations of applicability and specificity into existential subtyping, we now describe the process of checking the validity of a set of overloaded declarations $\mathcal{D}_f$ according to the rules in Section 3.

We can check the No Duplicates Rule by verifying that for every pair of distinct function declarations $d_1, d_2 \in \mathcal{D}_f$ either $d_1 \not\preceq d_2$ or $d_1 \not\preceq d_2$.

The Meet Rule requires that every pair of declarations $d_1, d_2 \in \mathcal{D}_f$ has a meet in $\mathcal{D}_f$. Because the more specific relation on function declarations corresponds to the subtyping relation on the (existential) domain types, we just need to find a declaration $d_0 \in \mathcal{D}_f$ whose domain type $dom(d_0)$ is equivalent to the meet (under $\leq$) of the existential types $dom(d_1)$ and $dom(d_2)$. Figure 2 shows how to compute the meet of two existential types.

Lemma 7 $\delta_1 \wedge \delta_2$ (as defined in Figure 2) is the meet of δ_1 and δ_2 under $\leq$.

Proof: First we show that $\delta_1 \wedge \delta_2$ is the meet of δ_1 and δ_2 under $\lesssim$. That $\delta_1 \wedge \delta_2 \lesssim \delta_1$ and $\delta_1 \wedge \delta_2 \lesssim \delta_2$ is obvious. For any δ_0, if $\overline{U}$ and $\overline{V}$ are instantiations that prove $\delta_0 \lesssim \delta_1$ and $\delta_0 \lesssim \delta_2$, respectively, then we can use the instantiation $\overline{U}, \overline{V}$ to prove that $\delta_0 \leq \delta_1 \wedge \delta_2$.

Now we show that the meet under $\lesssim$ is also the meet under $\leq$. Suppose that $\delta_0 \leq \delta_1$, $\delta_0 \leq \delta_1$, and $\Delta \vdash \delta_0 \xrightarrow{\equiv} \delta_0'$. A little work lets us deduce that $\delta_0' \lesssim \delta_1 \wedge \delta_2$ and hence $\delta_0 \leq \delta_1 \wedge \delta_2$. The fact that $\delta_1 \wedge \delta_2 \leq \delta_1$ and $\delta_1 \wedge \delta_2 \leq \delta_2$ follows from the fact that $\lesssim$ implies $\leq$. □

We can check the Return Type Rule using the subtype relation on universal types.

Theorem 2 Let $d_1 = f[\![\Delta_1]\!]\, S_1 : T_1$ and $d_2 = f[\![\Delta_2]\!]\, S_2 : T_2$ be declarations in $\mathcal{D}_f$ with $d_1 \preceq d_2$. They satisfy the Return Type Rule if $arrow(d_1)$ is a subtype of the arrow type $\sigma_\wedge = \forall[\![\Delta_1, \Delta_2]\!](S_1 \cap S_2) \to T_2$.

Proof: Let $d_\wedge = f[\![\Delta_1, \Delta_2]\!] S_1 \cap S_2 : T_2$, so $arrow(d_\wedge) = \sigma_\wedge$. Note that $d_\wedge$ and d_1 are equally specific and that $d_\wedge$ and d_2 satisfy the Return Type Rule. Because $arrow(d_1) \leq \sigma_\wedge$, for every instance $U \to V$ of $\sigma_\wedge$ with $U \not\equiv \mathrm{Bottom}$, we can find an instance $U_1 \to V_1$ of $arrow(d_1)$ with $U <: U_1$ and $V_1 <: V$. Thus, the pair d_1 and d_2 satisfy the Return Type Rule because the pair $d_\wedge$, d_2 does. □

7. Constraint-Based Judgments

Up to this point the precise definitions of subtyping and exclusion between types (and quantifier reduction) have remained unspecified. In this section we describe a small language of type constraints and we define subtyping and exclusion with respect to constraints. Finally, with constraint-based subtyping and exclusion defined, we explain in more detail the notion of quantifier reduction used in the $\leq$ judgments (and thus in our rule-checking).

7.1 Inference Variables

Until now we have only considered types whose free variables are bound in an explicit type environment. To gather constraints, however, we must check subtype and exclusion relationships between types with unbound *inference* variables. Intuitively, we have no control over the constraints on a bound type variable (which are fixed by the associated type environment), but we may introduce constraints on an inference variable. While the syntax of type variables is uniform, we conventionally distinguish them by using the metavariables X and Y for bound type variables and I and J for inference type variables.

7.2 Judgment Forms

In Figure 3, we list the judgments for generating constraints. A judgment of the form $\Delta \vdash S * T \Leftarrow \mathcal{C}$ states that under the assumptions Δ, the constraint $\mathcal{C}$ on the inference variables implies the proposition $S * T$, where $*$ ranges over $<:$, $\not<:$, $\Diamond$, $\not\Diamond$, $\equiv$, and $\not\equiv$. If S and T contain no inference

Existential Meet: $\delta_1 \wedge \delta_2$

$$\left(\exists[\![\overline{X <: \{\overline{M}\}}]\!]T\right) \wedge \left(\exists[\![\overline{Y <: \{\overline{N}\}}]\!]U\right) \stackrel{\text{def}}{=} \exists[\![\overline{X <: \{\overline{M}\}}, \overline{Y <: \{\overline{N}\}}]\!](T \cap U)$$

$$\text{where } \overline{X} \cap \overline{Y} = \overline{X} \cap (FV(U) \cup FV(\overline{N})) = \overline{Y} \cap (FV(T) \cup FV(\overline{M})) = \emptyset$$

Figure 2. The computed meet of existential types.

Primitive Judgments: $\Delta \vdash T * T \Leftarrow \mathcal{C}$

$$\Delta \vdash S <: T \Leftarrow \mathcal{C} \qquad \Delta \vdash S \Diamond T \Leftarrow \mathcal{C}$$
$$\Delta \vdash S \not<: T \Leftarrow \mathcal{C} \qquad \Delta \vdash S \not\Diamond T \Leftarrow \mathcal{C}$$

Derived Judgments: $\Delta \vdash T * T \Leftarrow \mathcal{C}$

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

$$\frac{\Delta \vdash S \not<: T \Leftarrow \mathcal{C} \qquad \Delta \vdash T \not<: S \Leftarrow \mathcal{C}'}{\Delta \vdash S \not\equiv T \Leftarrow \mathcal{C} \vee \mathcal{C}'}$$

Derived Judgments: $\Delta \vdash T * T \Rightarrow \mathcal{C}$

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

Figure 3. Constraint-based judgment forms.

variables the judgment behaves like an unconditional judgment (i.e., it only produces the constraints true or false).

Similarly, the judgment of the form

$$\Delta \vdash S * T \Rightarrow \mathcal{C}$$

states that under the assumptions Δ, if the proposition $S * T$ holds, then $\mathcal{C}$ must be true of the inference variables. In particular, when $\mathcal{C}$ holds of the inference variables, $S * T$ does not have to hold for every valid instantiation of the bound type variables. (Note that we only make use of this judgment where $*$ is $\not\equiv$.)

An important point about both kinds of judgments is that the types S and T should be considered *inputs* and the constraint $\mathcal{C}$ should be considered an *output*.

7.3 Constraint Forms

Our grammar for type constraints is defined in Figure 4. A primitive constraint is either *positive* or *negative*. We define positive primitive constraints: $S <: T$ specifies that a S is a subtype of T, and $S \Diamond T$ specifies that S must exclude T. Similarly, we define negative primitive constraints: $S \not<: T$ and $S \not\Diamond T$ with the obvious interpretations. A conjunction

Constraint Grammar

$$\begin{array}{rcl} \mathcal{C} & ::= & S <: T \\ & | & S \Diamond T \\ & | & S \not<: T \\ & | & S \not\Diamond T \\ & | & \mathcal{C} \wedge \mathcal{C} \\ & | & \mathcal{C} \vee \mathcal{C} \\ & | & \text{false} \\ & | & \text{true} \end{array}$$

Constraint Utilities

$$\neg\mathcal{C} = \mathcal{C}$$

$$\mathit{toConstraint}(\Delta) = \mathcal{C}$$

$$\mathit{toBounds}(\mathcal{C}) = \Delta$$

$$\Delta \vdash \mathit{unify}(\mathcal{C}) = \phi,\ \mathcal{C}$$

Figure 4. Constraints. Note that *unify* and *toBounds* are partial functions.

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. The equivalence constraint $S \equiv T$ is derived as $S <: T \wedge T <: S$.

Following Smith and Cartwright [16], we normalize all constraint formulas into disjunctive normal form and simplify away obvious contradictions and redundancies. We further make use of some auxiliary meta-level definitions, defined in Figure 4. The negation $\neg\mathcal{C}$ of a constraint $\mathcal{C}$ has a standard de Morgan interpretation. Each type environment $\Delta = \overline{X <: \{\overline{M}\}}$ naturally describes a constraint on the variables $\overline{X}$, which we denote *toConstraint*(Δ). This conversion has a partial inverse *toBound*$(\mathcal{C})$ that is defined whenever $\mathcal{C}$ can be written as a conjunction of constraints of the form $X <: M$.[13]

7.4 Subtyping

Figure 5 presents the full definition of our constraint-based subtyping algorithm as inference rules for the judgment $\Delta \vdash T <: T \Leftarrow \mathcal{C}$. Note that our algorithm requires that these rules be processed in the given order.

Smith and Cartwright similarly define a sound and complete algorithm for generating constraints from the Java subtyping relation [16]. We essentially preserve their semantics with two notable differences. First, our definition includes additional rules for tuple types to account for the fact that

[13] If $\mathcal{C}$ has multiple conjuncts of this form for a single X, then the resulting environment contains multiple bounds for X using the $\{\overline{M}\}$ notation.

Subtyping Rules: $\Delta \vdash T <: T \Leftarrow \mathcal{C}$

Logical rules

$$\frac{}{\Delta \vdash \text{Bottom} <: T \Leftarrow \text{true}}$$

$$\frac{}{\Delta \vdash M <: \text{Bottom} \Leftarrow \text{false}}$$

$$\frac{}{\Delta \vdash S \to T <: \text{Bottom} \Leftarrow \text{false}}$$

$$\frac{}{\Delta \vdash T <: \text{Any} \Leftarrow \text{true}}$$

$$\frac{}{\Delta \vdash \text{Any} <: S \to T \Leftarrow \text{false}}$$

$$\frac{\Delta \vdash T <: U \Leftarrow \mathcal{C} \qquad \Delta \vdash T <: V \Leftarrow \mathcal{C}'}{\Delta \vdash T <: U \cap V \Leftarrow \mathcal{C} \wedge \mathcal{C}'}$$

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

$$\frac{\Delta \vdash T <: U \Leftarrow \mathcal{C} \qquad \Delta \vdash T <: V \Leftarrow \mathcal{C}'}{\Delta \vdash T <: U \cup V \Leftarrow \mathcal{C} \vee \mathcal{C}'}$$

$$\frac{\Delta \vdash S <: U \Leftarrow \mathcal{C} \qquad \Delta \vdash T <: U \Leftarrow \mathcal{C}'}{\Delta \vdash S \cup T <: U \Leftarrow \mathcal{C} \wedge \mathcal{C}'}$$

Inference Variables

$$\frac{I \notin \Delta}{\Delta \vdash I <: I \Leftarrow \text{true}}$$

$$\frac{I \notin \Delta}{\Delta \vdash I <: T \Leftarrow I <: T}$$

$$\frac{I \notin \Delta}{\Delta \vdash S <: I \Leftarrow S <: I}$$

Bound Variables

$$\frac{}{\Delta \vdash X <: X \Leftarrow \text{true}}$$

$$\frac{\Delta \vdash \Delta(X) <: T \Leftarrow \mathcal{C}}{\Delta \vdash X <: T \Leftarrow \mathcal{C}}$$

$$\frac{\Delta \vdash S <: \text{Bottom} \Leftarrow \mathcal{C}}{\Delta \vdash S <: X \Leftarrow \mathcal{C}}$$

Structural rules

$$\frac{|\overline{S}| = |\overline{T}| \qquad \forall i. \quad \Delta \vdash S_i <: T_i \Leftarrow \mathcal{C}_i \qquad \forall i. \quad \Delta \vdash S_i <: \text{Bottom} \Leftarrow \mathcal{D}_i}{\Delta \vdash (\overline{S}) <: (\overline{T}) \Leftarrow (\bigwedge \mathcal{C}_i) \vee (\bigvee \mathcal{D}_i)}$$

$$\frac{|\overline{S}| \neq 1 \qquad \forall i. \quad \Delta \vdash S_i <: \text{Bottom} \Leftarrow \mathcal{C}_i}{\Delta \vdash (\overline{S}) <: T \Leftarrow \bigvee \mathcal{C}_i}$$

$$\frac{\Delta \vdash U <: S \Leftarrow \mathcal{C} \qquad \Delta \vdash T <: V \Leftarrow \mathcal{C}'}{\Delta \vdash S \to T <: U \to V \Leftarrow \mathcal{C} \wedge \mathcal{C}'}$$

$$\frac{|\overline{U}| \neq 1}{\Delta \vdash S \to T <: (\overline{U}) \Leftarrow \text{false}}$$

$$\frac{|\overline{T}| \neq 1}{\Delta \vdash M <: (\overline{T}) \Leftarrow \text{false}}$$

$$\frac{}{\Delta \vdash S \to T <: M \Leftarrow \text{false}}$$

$$\frac{}{\Delta \vdash M <: S \to T \Leftarrow \text{false}}$$

Constructed types

$$\frac{C \neq D \qquad \forall M \in C[\![\overline{S}]\!].\mathit{extends}. \quad \Delta \vdash M <: D[\![\overline{T}]\!] \Leftarrow \mathcal{C}_M}{\Delta \vdash C[\![\overline{S}]\!] <: D[\![\overline{T}]\!] \Leftarrow \bigvee \mathcal{C}_M}$$

$$\frac{\forall i. \quad \Delta \vdash S_i \equiv T_i \Leftarrow C_i}{\Delta \vdash C[\![\overline{S}]\!] <: C[\![\overline{T}]\!] \Leftarrow \bigwedge C_i}$$

Figure 5. Algorithm for generating subtyping constraints. Apply the first rule that matches.

any tuple is equivalent to Bottom if any of its element types is equivalent to Bottom.

$$\frac{|\overline{S}| = |\overline{T}| \qquad \forall i.\ \ \Delta \vdash S_i <: T_i \Leftarrow \mathcal{C}_i \qquad \forall i.\ \ \Delta \vdash S_i <: \mathrm{Bottom} \Leftarrow \mathcal{C}'_i}{\Delta \vdash (\overline{S}) <: (\overline{T}) \Leftarrow (\bigwedge \mathcal{C}_i) \vee (\bigvee \mathcal{C}'_i)}$$

$$\frac{|\overline{S}| \neq 1 \qquad \forall i.\ \ \Delta \vdash S_i <: \mathrm{Bottom} \Leftarrow \mathcal{C}_i}{\Delta \vdash (\overline{S}) <: T \Leftarrow \bigvee \mathcal{C}_i}$$

Second, our definition includes an additional rule for intersection types to account for exclusion since the intersection of excluding types is equivalent to Bottom. This rule makes our exclusion and subtyping rules mutually dependent.

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

We formally define the subtyping judgment from Section 2 as a trivial application of constraint-based subtyping with the following rule:

$$\frac{\Delta \vdash S <: T \Leftarrow \text{true}}{\Delta \vdash S <: T}$$

7.5 Exclusion

Figure 6 presents our definition of constraint-based exclusion as inference rules for the judgment $\Delta \vdash T \lozenge T \Leftarrow \mathcal{C}$. As with subtyping, our algorithm requires that these rules be processed in order. Additionally, if no rule matches, then the l.h.s. and r.h.s. types should be swapped and the rules tried again.

To make these rules algorithmic, we break the exclusion relation on constructed types into four subrelations $\lozenge_{\mathrm{x}}$, $\lozenge_{\mathrm{c}}$, $\lozenge_{\mathrm{o}}$, and $\lozenge_{\mathrm{m}}$. The first three relations are further decomposed into the asymmetric relations $\triangleright_{\mathrm{x}}$, $\triangleright_{\mathrm{c}}$, and $\triangleright_{\mathrm{o}}$.

1. $C[\![\,\overline{S}\,]\!] \triangleright_{\mathrm{x}} D[\![\,\overline{T}\,]\!]$ determines whether $D[\![\,\overline{T}\,]\!]$ has a super type N such that N appears in the excludes clause of an ancestor of $C[\![\,\overline{S}\,]\!]$.
2. $C[\![\,\overline{S}\,]\!] \triangleright_{\mathrm{c}} D[\![\,\overline{T}\,]\!]$ determines whether $D[\![\,\overline{T}\,]\!]$ excludes every type in the (nontrivial) comprises clause of $C[\![\,\overline{S}\,]\!]$.
3. $C[\![\,\overline{S}\,]\!] \triangleright_{\mathrm{o}} D[\![\,\overline{T}\,]\!]$ determines whether $C[\![\,\overline{S}\,]\!]$ is an object and $D[\![\,\overline{T}\,]\!]$ is not a supertype of $C[\![\,\overline{S}\,]\!]$.
4. $C[\![\,\overline{S}\,]\!] \lozenge_{\mathrm{m}} D[\![\,\overline{T}\,]\!]$ determines whether there is a pair of types (M, N) such that M is an ancestor of $C[\![\,\overline{S}\,]\!]$, N is an ancestor of $D[\![\,\overline{T}\,]\!]$, and M and N are distinct applications of the same type constructor.

As with subtyping, we formally define the exclusion judgment described in Section 4 as a trivial application of constraint-based exclusion with the following rule:

$$\frac{\Delta \vdash S \lozenge T \Leftarrow \text{true}}{\Delta \vdash S \lozenge T}$$

7.6 Negative Judgments and Negation

In the rules for constraint-based exclusion (Figure 6), we use the negative judgments $\Delta \vdash T \not<: T \Leftarrow \mathcal{C}$ and $\Delta \vdash T \not\lozenge T \Leftarrow \mathcal{C}$ to determine constraints under which the negations hold. Instead of defining negative judgments explicitly, we describe how to derive them from their positive counterparts according to de Morgan's laws.

For the negative subtyping judgment $\Delta \vdash T \not<: T \Leftarrow \mathcal{C}$, the rules for bound variables are given below:

$$\frac{X \in \Delta}{\Delta \vdash X \not<: T \Leftarrow \text{false}} \qquad \frac{\Delta \vdash S \not<: \Delta(X) \Leftarrow \mathcal{C}}{\Delta \vdash S \not<: X \Leftarrow \mathcal{C}}$$

The other rules for $\Delta \vdash T \not<: T \Leftarrow \mathcal{C}$ are obtained as follows: For each rule of $\Delta \vdash T <: T \Leftarrow \mathcal{C}$ that is not in the section marked "bound variables," make a new rule for $\Delta \vdash T \not<: T \Leftarrow \mathcal{C}$ by replacing each occurrence of a relation symbol $*$ with its negation, and by swapping each $\wedge$ with $\vee$ and true with false, and vice versa. For example, the rule for intersection types on the r.h.s.

$$\frac{\Delta \vdash T <: U \Leftarrow \mathcal{C} \qquad \Delta \vdash T <: V \Leftarrow \mathcal{C}'}{\Delta \vdash T <: U \cap V \Leftarrow \mathcal{C} \wedge \mathcal{C}'}$$

becomes the following rule in the negative subtyping judgment

$$\frac{\Delta \vdash T \not<: U \Leftarrow \mathcal{C} \qquad \Delta \vdash T \not<: V \Leftarrow \mathcal{C}'}{\Delta \vdash T \not<: U \cap V \Leftarrow \mathcal{C} \vee \mathcal{C}'}$$

The rules for the negative exclusion judgment $\Delta \vdash T \not\lozenge T \Leftarrow \mathcal{C}$ are derived from those of $\Delta \vdash T \lozenge T \Leftarrow \mathcal{C}$ according to the process above. The rule for bound variables is

$$\frac{X \in \Delta}{\Delta \vdash X \not\lozenge T \Leftarrow \text{false}}$$

The negative subtyping judgment should not be confused with the derived contrapositive judgment $\Delta \vdash T \not\equiv T \Rightarrow \mathcal{C}$ given in Figure 3, for the two judgments handle bound type variables very differently. Intuitively, the negative assertion $\Delta \vdash S \not\equiv T \Leftarrow \mathcal{C}$ computes the constraint $\mathcal{C}$ that satisfies the inequivalence for an arbitrary instantiation of the type variables bound in Δ. Whereas the contrapositive assertion $\Delta \vdash S \not\equiv T \Rightarrow \mathcal{C}$ computes the constraint $\mathcal{C}$ that holds for any instantiation of Δ such that the inequivalence is true. The following derivable assertions further illustrate this distinction, for $\Delta = X <: \mathrm{Any}, Y <: \mathrm{Any}$:

$$\Delta \vdash \big(\mathrm{Pair}[\![X, I]\!] \cap \mathrm{Pair}[\![Y, J]\!]\big) \not\equiv \mathrm{Bottom} \Leftarrow \text{false}$$
$$\Delta \vdash \big(\mathrm{Pair}[\![X, I]\!] \cap \mathrm{Pair}[\![Y, J]\!]\big) \not\equiv \mathrm{Bottom} \Rightarrow I \equiv J$$

7.7 Unification

Suppose that $\mathcal{C}$ is a conjunction of type equivalences. A *unifier* of $\mathcal{C}$ is a substitution ϕ of types for inference type

Exclusion: $\Delta \vdash T \lozenge T \Leftarrow \mathcal{C}$

Logical rules

$$\frac{}{\Delta \vdash \text{Bottom} \lozenge T \Leftarrow \text{true}}$$

$$\frac{\Delta \vdash T <: \text{Bottom} \Leftarrow \mathcal{C}}{\Delta \vdash \text{Any} \lozenge T \Leftarrow \mathcal{C}}$$

$$\frac{\Delta \vdash S \lozenge U \Leftarrow \mathcal{C} \qquad \Delta \vdash T \lozenge U \Leftarrow \mathcal{C}' \qquad \Delta \vdash S \cap T <: \text{Bottom} \Leftarrow \mathcal{C}''}{\Delta \vdash S \cap T \lozenge U \Leftarrow \mathcal{C} \vee \mathcal{C}' \vee \mathcal{C}''}$$

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

Inference Variables

$$\frac{I \notin \Delta}{\Delta \vdash I \lozenge I \Leftarrow \text{false}}$$

$$\frac{I \notin \Delta}{\Delta \vdash I \lozenge T \Leftarrow I \lozenge T}$$

Bound Variables

$$\frac{\Delta \vdash \Delta(X) \lozenge T \Leftarrow C}{\Delta \vdash X \lozenge T \Leftarrow C}$$

Structural rules

$$\frac{|\overline{S}| = |\overline{T}| \qquad \forall i. \Delta \vdash S_i \lozenge T_i \Leftarrow \mathcal{C}_i}{\Delta \vdash (\overline{S}) \lozenge (\overline{T}) \Leftarrow \bigvee \mathcal{C}_i}$$

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

$$\frac{M \neq \text{Any} \qquad |\overline{T}| \neq 1}{\Delta \vdash M \lozenge (\overline{T}) \Leftarrow \text{true}}$$

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

$$\frac{}{\Delta \vdash S \to T \lozenge U \to V \Leftarrow \text{false}}$$

$$\frac{M \neq \text{Any}}{\Delta \vdash M \lozenge T \to U \Leftarrow \text{true}}$$

Constructed types

$$\frac{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\text{x}} D[\![\overline{T}]\!] \Leftarrow \mathcal{C}_e \quad \Delta \vdash C[\![\overline{S}]\!] \lozenge_{\text{c}} D[\![\overline{T}]\!] \Leftarrow \mathcal{C}_c \quad \Delta \vdash C[\![\overline{S}]\!] \lozenge_{\text{o}} D[\![\overline{T}]\!] \Leftarrow \mathcal{C}_o \quad \Delta \vdash C[\![\overline{S}]\!] \lozenge_{\text{m}} D[\![\overline{T}]\!] \Leftarrow \mathcal{C}_p}{\Delta \vdash C[\![\overline{S}]\!] \lozenge D[\![\overline{T}]\!] \Leftarrow \mathcal{C}_e \vee \mathcal{C}_c \vee \mathcal{C}_o \vee \mathcal{C}_p}$$

$$\frac{\Delta \vdash C[\![\overline{S}]\!] \rhd_* D[\![\overline{T}]\!] \Leftarrow \mathcal{C} \quad \Delta \vdash D[\![\overline{T}]\!] \rhd_* C[\![\overline{S}]\!] \Leftarrow \mathcal{C}'}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_* D[\![\overline{T}]\!] \Leftarrow \mathcal{C} \vee \mathcal{C}'} \quad \text{where } * \in \{\text{x}, \text{c}, \text{o}\}$$

$$\frac{}{\Delta \vdash C[\![\overline{S}]\!] \rhd_* C[\![\overline{T}]\!] \Leftarrow \text{false}} \quad \text{where } * \in \{\text{x}, \text{c}, \text{o}\}$$

$$\frac{C \neq D \qquad A = \mathit{ancestors}(C[\![\overline{S}]\!]) \qquad \forall N \in \left(\bigcup_{M \in A} M.\mathit{excludes}\right). \quad \Delta \vdash D[\![\overline{T}]\!] <: N \Leftarrow \mathcal{C}_N}{\Delta \vdash C[\![\overline{S}]\!] \rhd_{\text{x}} D[\![\overline{T}]\!] \Leftarrow \bigvee \mathcal{C}_N}$$

$$\frac{C \neq D \qquad \forall M \in C[\![\overline{S}]\!].\mathit{comprises}. \quad \Delta \vdash M \lozenge D[\![\overline{T}]\!] \Leftarrow \mathcal{C}_M}{\Delta \vdash C[\![\overline{S}]\!] \rhd_{\text{c}} D[\![\overline{T}]\!] \Leftarrow \bigwedge \mathcal{C}_M}$$

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

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

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

$$\frac{\forall M \in \mathit{ancestors}(C[\![\overline{S}]\!]). \quad \forall N \in \mathit{ancestors}(D[\![\overline{T}]\!]). \quad \Delta \vdash M \lozenge_{\text{m}} N \Leftarrow \mathcal{C}_{M,N}}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\text{m}} D[\![\overline{T}]\!] \Leftarrow \bigvee \mathcal{C}_{M,N}}$$

$$\frac{\forall i. \Delta \vdash S_i \not\equiv T_i \Leftarrow \mathcal{C}_i}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\text{m}} C[\![\overline{T}]\!] \Leftarrow \bigvee \mathcal{C}_i}$$

$$\frac{C \neq D}{\Delta \vdash C[\![\overline{S}]\!] \lozenge_{\text{m}} D[\![\overline{T}]\!] \Leftarrow \text{false}}$$

Figure 6. Algorithm for generating exclusion constraints. Each rule is symmetric; apply the first one that matches.

variables such that $\phi(\mathcal{C}) = \text{true}$. We say that a unifier ϕ is more general than a unifier ψ if there exists a substitution τ such that $\tau \circ \phi = \psi$. In other words ϕ is more general than ψ if ψ factors throughout ϕ.

We can extend the notion of a unifier to an arbitrary conjunction $\mathcal{C}$ in the case that $\mathcal{C}$ can be expressed as a conjunction $\mathcal{C}' \wedge \mathcal{C}''$ where $\mathcal{C}'$ is entirely equivalences and $\mathcal{C}''$ contains no type equivalences. Then we define a unifier of $\mathcal{C}$ to be a unifier of $\mathcal{C}'$. Finally, we can extend the notion of unifier to a constraint $\mathcal{C}$ in disjunctive normal form to be a unifier of any disjunct of $\mathcal{C}$.

The (partial) function *unify* in Figure 4 takes a constraint $\mathcal{C}$ and produces a most general unifier ϕ if one exists. This is always the case if $\mathcal{C}$ consists of a single conjunct. *unify* additionally produces the substituted leftover part, $\phi(\mathcal{C}'')$.

7.8 Quantifier Reduction

In the evaluation of valid overloadings from Section 5, intensional type analysis was required in order to reason about certain examples. Since this reasoning justified the validity of these overloaded functions, we incorporate it into the present formal system as well.

Whenever two different domain types should be applicable to the same argument type W (in order to validate the Meet Rule or Return Type Rule), an existentially quantified intersection type naturally arises as the necessary supertype of W. Intersection types $S \cap T$ in our type system naturally fall into two distinct cases: either $S \not\Diamond T$, or $S \Diamond T$ in which case the intersection has the same extent as Bottom. In the second case, the intersection is trivial and W, as a subtype of the intersection, must also be trivial. Moreover, because the argument type W to which both declarations must be applicable is necessarily equivalent to Bottom, then the Meet Rule and Return Type Rule are both trivially satisfied by the presence of the implicit overloading on Bottom. In this manner case analysis on whether an existentially quantified (intersection) type is Bottomfacilitates the checking of our rules.

Naïvely one might expect this case analysis on $S \cap T$ to simply check whether $S \Diamond T$. However, as is the case when checking generic function declarations, the types S and T might have free type variables, whose uncertainty often precludes a definitive statement about $S \Diamond T$. (For example, $C[\![X]\!] \Diamond C[\![Y]\!]$ holds only if $X \not\equiv Y$.) Our solution is to reason backwards: Under the assumption that the intersection is nontrivial (that the types do not exclude), gather the necessary constraints on type parameters. (For example, $C[\![X]\!] \cap C[\![Y]\!] \not\equiv \mathrm{Bottom}$ yields the constraint $X \equiv Y$.) These constraints are then reduced, resulting in an instantiation (and potentially tighter bounds on type parameters) that necessarily follows from our assumption of nontriviality.[14]

We call the general pattern of simplifying an existentially quantified (intersection) type *existential reduction*, given by the judgment $\Delta \vdash \delta \xrightarrow{\equiv} \delta$ in Figure 7. The first rule for existential reduction performs the constraint-based case analysis described above, while the second merely relates the existential to itself if the premises of the first rule do not hold. We thus explain the first rule in more detail.

The first premise determines the constraints $\mathcal{C}$ that must be true under the hypothesis that $T \not\equiv \mathrm{Bottom}$ (i.e. that this type is nontrivial). Note that the type variables from Δ are bound, while any type variables from the existential itself, Δ', become inference type variables mentioned in $\mathcal{C}$. The second premise binds $\mathcal{C}'$ to exactly the inference type variables and bounds denoted by the existential's type parameters; these are the constraints that must hold for T to still make sense. In the third premise, if *unify* succeeds, it produces a substitution ϕ for any inference type variables from Δ' constrained by equalities. Because ϕ is a most general unifier, it has the property that any other valid substitution ψ of Δ''s variables with $\psi(T) \not\equiv \mathrm{Bottom}$ must be equal to $\tau \circ \phi$, for some other substitution τ. Moreover, if *unify* succeeds, it produces a set of leftover constraints $\mathcal{C}''$ that are not unifiable equalities (but have still been simplified). If it is possible to express $\mathcal{C}''$ as some type environment Δ'', then we use this as the new type parameters over the simplified type $\phi(T)$.

Similarly we call the general pattern of simplifying a universally quantified arrow type *universal reduction*, given by the judgment $\Delta \vdash \sigma_1 \xrightarrow{\equiv} \sigma_2$ The first premise reduces the domain type $dom(\sigma) = \exists[\![\Delta']\!]S$, resulting in a new existential type $\delta = \exists[\![\Delta'']\!]S'$ and a substitution ϕ mapping type variables from Δ' to types with variables in Δ''. We then construct a new arrow with domain δ and range $\phi(T)$.

As an example, in order to check that the first two declarations of $\mathcal{D}_{bar}$ from Section 5 satisfy the Meet Rule, we must reduce the existential

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

Thus we must find the constraint $\mathcal{C}$ such that

$$\vdash \mathrm{ArrayList}[\![X]\!] \cap \mathrm{List}[\![Y]\!] \not\equiv \mathrm{Bottom} \Rightarrow \mathcal{C}$$

can be derived, noting that X and Y are actually (unbound) type inference variables here. In this instance $\mathcal{C} = X \equiv Y$ due to multiple instantiation exclusion. Then we convert the bounds on the existential's type parameters into the constraint $\mathcal{C}'$ on X and Y as inference variables: *toConstraint*$(X <: \mathrm{Any}, Y <: \mathbb{Z}) = X <: \mathrm{Any}, Y <: \mathbb{Z}$. Unifying the constraint

$$\mathcal{C} \wedge \mathcal{C}' = X \equiv Y \wedge X <: \mathrm{Any} \wedge Y <: \mathbb{Z}$$

yields the type substitution $\phi = [W/X, W/Y]$ (for some fresh variable W) and the simplified leftover constraint $\mathcal{C}'' =$

[14] A similar sort of case analysis and constraint solving arises for pattern matching with generalized algebraic data types (GADTs) [15]: GADTs resemble our existential types and pattern matching resembles our function application.

$$\boxed{\textbf{Existential Reduction:} \quad \Delta \vdash \delta \xrightarrow{\equiv} \delta\,,\ \phi}$$

$$\frac{\Delta \vdash T \not\equiv \text{Bottom} \Rightarrow \mathcal{C} \qquad \textit{toConstraint}(\Delta') = \mathcal{C}' \quad \Delta \vdash \textit{unify}(\mathcal{C} \wedge \mathcal{C}') = \phi\,,\ \mathcal{C}'' \qquad \textit{toBounds}(\mathcal{C}'') = \Delta''}{\Delta \vdash \exists[\![\Delta']\!]T \xrightarrow{\equiv} \exists[\![\Delta'']\!]\phi(T)\,,\ \phi}$$

$$\frac{\text{otherwise}}{\Delta \vdash \exists[\![\Delta']\!]T \xrightarrow{\equiv} \exists[\![\Delta']\!]T\,,\ [/]}$$

$$\boxed{\textbf{Universal Reduction:} \quad \Delta \vdash \sigma \xrightarrow{\equiv} \sigma\,,\ \phi}$$

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

Figure 7. Quantifier reduction judgments.

$W <: \mathbb{Z}$. Since $\mathcal{C}''$ has the form of a type environment, $\textit{toBounds}(\mathcal{C}'') = W <: \mathbb{Z}$, we finally reduce this existential to $\exists[\![W <: \mathbb{Z}]\!]\big(\text{ArrayList}[\![W]\!] \cap \text{List}[\![W]\!]\big)$. However, due to the class table declaration of $\text{ArrayList}[\![W]\!]$ this existential type will be indistinguishable (by $\lesssim$) from the simpler $\exists[\![W <: \mathbb{Z}]\!]\text{ArrayList}[\![W]\!]$.

When checking that the first two declarations $\mathcal{D}_{tail}$ from Section 5 satisfy the Return Type Rule, we use universal reduction to prove

$$\begin{array}{ll} \vdash & \forall[\![X <: \text{Any}]\!]\text{List}[\![X]\!] \to \text{List}[\![X]\!] \\ \leq & \forall[\![X <: \text{Any}, Y <: \text{Number}]\!]\big(\text{List}[\![X]\!] \cap \text{List}[\![Y]\!]\big) \\ & \qquad\qquad \to \text{List}[\![Y]\!] \end{array}$$

First, note that by reasoning similar to that in the previous example we know

$$\begin{array}{l} \vdash \exists[\![X <: \text{Any}, Y <: \text{Number}]\!]\big(\text{List}[\![X]\!] \cap \text{List}[\![Y]\!]\big) \\ \qquad \xrightarrow{\equiv} \exists[\![W <: \text{Number}]\!]\text{List}[\![W]\!] \end{array}$$

with substitution $[W/X, W/Y]$. Using this substitution we must now prove

$$\begin{array}{ll} \vdash & \forall[\![X <: \text{Number}]\!]\text{List}[\![X]\!] \to \text{List}[\![X]\!] \\ \lesssim & \forall[\![W <: \text{Number}]\!]\text{List}[\![W]\!] \to \text{List}[\![W]\!] \end{array}$$

which is clearly true.

8. Overloading Across Modules

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 safety 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 are the unions 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:[15] Suppose we have a type Number in module A, with a function:

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

Suppose we have the type and function:

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

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, one 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.

Castagna *et al.* proposed rules for defining overloaded functions to ensure type safety [4]. They 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).

Millstein and Chambers devised the language *Dubious* to study how to modularly ensure safety for overloaded functions with symmetric multiple dynamic dispatch (*multimethods*) in a type system supporting multiple inheritance [11, 12]. With Clifton and Leavens, they developed MultiJava [5], an extension of Java with Dubious' semantics for multimethods. Lee and Chambers presented F(EML) [9], a language with classes, symmetric multiple dispatch, and parameterized modules. In previous work [2], we built on the work of Millstein and Chambers to give modular rules for a core of the Fortress 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), but defines an *exclusion relation* on types to allow more valid overloadings. For detailed comparison of modularity and dispatch for these systems, see the related work section of our previous paper [2].

None of the systems described above support polymorphic functions or types. F(EML)'s parameterized modules (*functors*) provide a form of parametricity but they cannot be implicitly applied; the functions defined therein cannot be *overloaded* with those defined in other functors. This paper extends our previous work [2] with parametric polymorphism for both types and top-level functions.

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 greatly simplifies the checking of their equivalent of the Return Type Rule: the return type of each definition needs to be compared only with the return type of the principal type. Also, to check their equivalent of the Meet Rule, the entire type hierarchy relevant to the multimethod must be known, so in general, this check must be done at link time. Finally, their type system does not provide any exclusion relation. On the other hand, $\text{ML}_{\leq}$ allows variance annotations on type constructors—something we defer 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 [17] introduced the *type class* as a means to specify and implement overloaded functions such as equality and arithmetic operators in Haskell. Other authors have translated type classes to languages besides Haskell [6, 14, 18]. 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:

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

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

or functions lacking uniform variance in the domain and range[16] like:

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

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 [17] 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 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 with multiple inheritance, richer type relations must be available to programmers—the subtyping relation prevalent among such languages does not afford enough type analysis alone. We have therefore 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, 13]. They add additional complexity to polymorphic exclusion checking, so we leave them to future work.

Acknowledgments

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

[16] 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`.

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*, 2007.

[3] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *POPL '97*, 1997.

[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 TOPLAS*, 28(3), 2006.

[6] Derek Dreyer, Robert Harper, and Manuel M. T. Chakravarty. Modular type classes. In *POPL '07*, 2007.

[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*, 2006.

[10] Vassily Litvinov. Constraint-based polymorphism in Cecil: Towards a practical and static type system. In *OOPSLA '98*, 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] Martin Odersky. *The Scala Language Specification, Version 2.7*. EPFL Lausanne, Switzerland, 2009.

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

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

[16] Daniel Smith and Robert Cartwright. Java type inference is broken: can we fix it? In *OOPSLA '08*, 2008.

[17] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89*, 1989.

[18] Stefan Wehr, Ralf Lämmel, and Peter Thiemann. JavaGI: Generalized interfaces for Java. In *ECOOP 2007*, 2007.