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

Eric Allen
Sun Labs, Oracle
eric.allen@oracle.com

Justin Hilburn
Sun Labs, Oracle
justin.hilburn@oracle.com

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

Sukyoung Ryu
Korea Advanced Institute of Science and Technology
sryu@cs.kaist.ac.kr

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

Victor Luchangco
Sun Labs, Oracle
victor.luchangco@oracle.com

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

Abstract

Multiple dynamic dispatch poses many problems for a statically typed language with nominal subtyping and multiple inheritance. In particular, as Millstein *et al.* have observed, there is a tension between modularity and extensibility when trying to ensure at compile time that each function call results in dispatch to a unique most specific definition at run time. We have previously shown how requiring that each set of overloaded definitions forms a meet semi-lattice, as suggested in 1992 by Castagna *et al.*, allows one to statically ensure the absence of ambiguous function calls while maintaining modularity and extensibility.

In this paper, we extend the rules for ensuring safe overloaded functions to a type system with parametric polymorphism. We show that such an extension can be reduced to the problem of determining subtyping relationships between universal and existential types. However, because of interactions between multiple inheritance, parametric types, type inference, and statically determined exclusiveness of types, there are syntactically distinct types inhabited by the same sets of values. Work must be done to ensure that they are equivalent under subtyping.

Our system has been adopted by the Fortress programming language, which includes support for modularity, multiple inheritance, mulitple dispatch, and parametric polymorphism. We have implemented this design as part of the freely available open source Fortress compiler.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]

General Terms Programming languages

Keywords object-oriented programming, multiple dispatch, symmetric dispatch, multiple inheritance, overloading, modularity, multimethods, static types, components, separate compilation, parametric polymorphism, ad-hoc polymorphism

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, and calls to a function (method) of that name are resolved based on the "runtime types" (we use the term *ilks*) of some arguments. In *single dispatch*, a particular argument is designated as the *receiver*, and the call is resolved only with respect to that argument. In *multiple dispatch*, the runtime types of all arguments to a function call are used to resolve the call. A special case of multiple dispatch is that of *symmetric multiple dispatch*, in which all arguments are considered equally when resolving a call.

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

To preserve type soundness while incorporating multiple dispatch into an object-oriented language with a static semantics, constraints must be placed on the sets of allowable overloaded definitions. For example, to avoid ambiguous function calls, we must ensure that for every call site (knowing only the static types of the arguments), there exists a unique "best" function to dispatch to at run time.[1]

Castagna *et al.* showed how to check overloaded function definitions to ensure this property in a language without parametric polymorphism [4]. In particular, the static type system must impose a partial order on each set of overloaded definitions. This partial order may be described as: m_1 is more specific than m_2 if and only if for every possible set of arguments, if m_1 is applicable then m_2 is also applicable. Every program is required to ensure that, for any two overloaded function definitions, if neither is more specific than the other, then there is a third overloaded definition that is (1) more specific than both and (2) no more specific than any other definable function that is more specific than both. We can think of this property as imposing a *meet semi-lattice* on a set of function declarations. Ensuring this property is complicated in a language with multiple dispatch because each of two function definitions might be more specific in one argument and not another. For example, consider the following overloaded function definitions:

$$f(b : B, a : A)$$
$$f(a : A, b : B)$$

[1] In languages with static overloading, such as Scala, C#, and the Java$^{\text{TM}}$ programming language [9], it is possible to simply reject ambiguous call sites of overloaded functions [9, 11, 17]. However, as Millstein and Chambers have observed, it is impossible to statically forbid ambiguity in the presence of multiple dynamic dispatch without imposing constraints at the definition sites of overloaded functions [14, 15].

POPL '11 September, Austin, Texas.
Copyright © 2010 ACM [to be supplied]. . . $10.00

If A is a subtype of B, to which of these definitions do we dispatch when f is called with two arguments of type A? Note that the ambiguity is inherent in these definitions: there is a real question as to what behavior the programmer intended in this case.

Resolving ambiguous function calls becomes even more important, and more difficult, in the presence of parametric polymorphism, where both types and functions can be parameterized by type variables. One way to think about a parametric type such as $\text{List}[\![T]\!]$ (a list with elements of type T) is that it represents an infinite set of ground types $\text{List}[\![\text{Object}]\!]$ (lists of objects), $\text{List}[\![\text{String}]\!]$ (lists of strings), $\text{List}[\![\mathbb{Z}]\!]$ (lists of integers), and so on. In an actual type checker, it is necessary to have rules for working with uninstantiated (non-ground) parametric types, but for many purposes this model of "an infinite set of ground types"' is adequate for explanatory purposes. Not so, however, for type-parametric functions. For quite some time during the development of the Fortress language, one of us (Steele) pushed for understanding a parametrically polymorphic function definition such as:

$$\mathit{append}[\![T]\!]\big(x\colon \text{List}[\![T]\!], y\colon \text{List}[\![T]\!]\big)\colon \text{List}[\![T]\!] = e$$

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

$$\begin{array}{l}\mathit{append}\big(x\colon \text{List}[\![\text{Object}]\!], y\colon \text{List}[\![\text{Object}]\!]\big)\colon \text{List}[\![\text{Object}]\!] = e\\ \mathit{append}\big(x\colon \text{List}[\![\text{String}]\!], y\colon \text{List}[\![\text{String}]\!]\big)\colon \text{List}[\![\text{String}]\!] = e\\ \mathit{append}\big(x\colon \text{List}[\![\mathbb{Z}]\!], y\colon \text{List}[\![\mathbb{Z}]\!]\big)\colon \text{List}[\![\mathbb{Z}]\!] = e\\ \dots\end{array}$$

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

That intuition worked well enough for a single polymorphic function definition, but failed utterly when we considered multiple function definitions. It would seem natural for a programmer to want to provide definitions for specific monomorphic special cases:

$$\begin{array}{l}\mathit{append}[\![T]\!]\big(x\colon \text{List}[\![T]\!], y\colon \text{List}[\![T]\!]\big)\colon \text{List}[\![T]\!] = e_1\\ \mathit{append}\big(x\colon \text{List}[\![\mathbb{Z}]\!], y\colon \text{List}[\![\mathbb{Z}]\!]\big)\colon \text{List}[\![\mathbb{Z}]\!] = e_2\end{array}$$

but if the model is taken seriously, this would be equivalent to:

$$\begin{array}{l}\mathit{append}\big(x\colon \text{List}[\![\text{Object}]\!], y\colon \text{List}[\![\text{Object}]\!]\big)\colon \text{List}[\![\text{Object}]\!] = e_1\\ \mathit{append}\big(x\colon \text{List}[\![\text{String}]\!], y\colon \text{List}[\![\text{String}]\!]\big)\colon \text{List}[\![\text{String}]\!] = e_1\\ \mathit{append}\big(x\colon \text{List}[\![\mathbb{Z}]\!], y\colon \text{List}[\![\mathbb{Z}]\!]\big)\colon \text{List}[\![\mathbb{Z}]\!] = e_1\\ \dots\\ \mathit{append}\big(x\colon \text{List}[\![\mathbb{Z}]\!], y\colon \text{List}[\![\mathbb{Z}]\!]\big)\colon \text{List}[\![\mathbb{Z}]\!] = e_2\end{array}$$

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

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

$$\begin{array}{l}\mathit{append}[\![T]\!]\big(x\colon \text{List}[\![T]\!], y\colon \text{List}[\![T]\!]\big)\colon \text{List}[\![T]\!] = e_1\\ \mathit{append}[\![T <\colon \text{Number}]\!]\big(x\colon \text{List}[\![T]\!], y\colon \text{List}[\![T]\!]\big)\colon \text{List}[\![T]\!] = e_2\end{array}$$

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

In order to resolve this problem, we had to develop an alternate model and an associated type system that could handle overloaded parametrically polymorphic methods in a manner that would accord with programmer intuition and support the plausible examples shown above. Credit for championing key insights—regarding each polymorphic definition as a single definition (rather than an infinite set of definitions) competing in the overload set, and using universal and existential types to describe them in the type system (an idea reported by Bourdoncle and Merz [3])—and for working out many difficult details belongs to two other authors of this paper (Hilburn and Kilpatrick). Adopting this new approach has made overloaded polymorphic functions (and methods) both tractable and effective for writing Fortress code.

In this paper, we give rules for ensuring safe overloaded functions in a language that supports symmetric multiple dispatch, multiple inheritance, and parametric polymorphism (that is, generic traits *and* generic functions). We do this by extending our earlier rules for a core of the Fortress programming language that did not support generics [1, 2]. This problem is challenging because a set of overloaded *generic* function definitions might have not only distinct argument types, but also distinct type parameters, so the type values of these parameters make sense only in distinct type environments. How do we compare such functions, and how do we ensure that there is always a best overloaded function to dispatch to?

For example, consider the following overloaded function definitions in Fortress:

$$\begin{array}{l}\mathit{combine}[\![T]\!]\big(\mathit{xs}\colon \text{List}[\![T]\!], \mathit{ys}\colon \text{List}[\![T]\!]\big)\colon \text{List}[\![T]\!]\\ \mathit{combine}[\![S,T]\!]\big(s\colon \text{Table}[\![S,T]\!], t\colon \text{Table}[\![S,T]\!]\big)\colon \text{Table}[\![S,T]\!]\end{array}$$

where white square brackets delimit declarations of type parameters on function definitions. The first definition declares a single type parameter denoting the types of the elements of the two list arguments xs and ys. The second definition declares two type parameters corresponding to the domains and ranges of the two table arguments s and t. But the type parameters of the first definition bear no relation to the type parameters of the second.

In providing rules to ensure that any 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. Our rules must also be compatible with type inference, since instantiation of type parameters at a call site is typically done automatically.

The remainder of this paper is organized thus: In Section 2, we define the concepts and notation necessary to explain our formal rules for checking overloaded function definitions, which we present using universal and existential types in Section 3. In Section 4, we explain why many natural programs are (rightfully) rejected by our rules and propose to extend the language with exclusion in order to allow them. In Section 5 we formally describe exclusion and use it to augment the subtyping relation for universal and existential types, Finally we discuss related work Section 6 and conclude in section 7.

2. Preliminaries

Following Kennedy and Pierce [12], we define a world of types ranged over by metavariables S, T, U, V, and W. Types are of four forms: Type variables (ranged over by metavariables X, Y, and Z), constructed types (ranged over by metavariables K, L, M and N, and written $C[\![\overline{T}]\!]$ where C is a type constructor and $\overline{T}$ is a list of types), structural types (consisting of arrow and tuple types), and compound types (consisting of union and intersection types). Nullary applications are written without brackets; for example, $C[\![\,]\!]$ is written C. In addition, there are two special constructed types, Any and BottomType, explained below. The abstract syntax of types is defined in BNF as follows (where $\overline{A}$ indicates a possibly empty sequence of syntactic elements A separated by commas):

$$\begin{array}{lcl} \mathit{Type} & ::= & \mathit{Id} \\ & \mid & \mathit{Id}[\![\overline{\mathit{Type}}]\!] \\ & \mid & \mathit{Type} \rightarrow \mathit{Type} \\ & \mid & (\overline{\mathit{Type}}) \\ & \mid & \mathit{Type} \cap \mathit{Type} \\ & \mid & \mathit{Type} \cup \mathit{Type} \end{array}$$

A tuple type of length one is synonymous with its element type. A tuple type with any $\mathrm{BottomType}$ element is synonymous with $\mathrm{BottomType}$. Compound types are *not* first-class in the Fortress language: they cannot be written in a program; rather, they are used by the type analyzer during type checking. However, 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.

Constructed types (other than Any and $\mathrm{BottomType}$) are applications of *type constructors*. A declaration for type constructor C has the following form:

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

indicating that an application $C[\![\overline{U}]\!]$ (i) is *well-formed* if and only if $U_i <: [\overline{U}/\overline{X}]M_{ij}$ for each bound M_{ij} (where $<:$ is the subtyping relation defined below, and $[\overline{U}/\overline{X}]M_{ij}$ is M_{ij} with U_k substituted for every occurrence of X_k in M_{ij} for $1 \leq k \leq |\overline{U}|$), and (ii) is a subtype of $[\overline{U}/\overline{X}]N_i$ for $1 \leq i \leq |\overline{N}|$. Thus, a set of type constructor declarations, called a *class table*, induces a (nominal) subtyping relation over the constructed types, by taking the reflexive and transitive closure of the relation derived from the declarations in the class table. In addition, every type is a subtype of Any and a supertype of $\mathrm{BottomType}$.

A class table is *well-formed* if the resulting subtyping relation on its constructed types is a partial order. As usual for languages with nominal subtyping, we allow recursive and mutually recursive references in the class table. Class tables are ranged over by metavariable $\mathcal{T}$ and we write $T \in \mathcal{T}$ to mean that any constructed type occurring in type T is well-formed with respect to the class table $\mathcal{T}$. A class table $\mathcal{T}'$ is an extension of another class table $\mathcal{T}$ (written $\mathcal{T}' \supseteq \mathcal{T}$) if every constructor declaration in $\mathcal{T}'$ is also in $\mathcal{T}$ and if the subtype relation on $\mathcal{T}'$ agrees with that of $\mathcal{T}$. Consequently, if $\mathcal{T}' \supseteq \mathcal{T}$ then, for all types $T, T \in \mathcal{T}$ implies $T \in \mathcal{T}'$.

For a well-formed application $C[\![\overline{T}]\!]$ of the type constructor declaration for C (shown above) in a well-formed class table $\mathcal{T}$, we denote the set of its immediate supertypes by

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

With this, we can derive the set of all ancestors of $C[\![\overline{T}]\!]$ with respect to $\mathcal{T}$ recursively:

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

We typically omit the class table when it is understood.

Structural and compound types are *well-formed* (with respect to a class table) if their constituent types are well-formed, and we extend the subtyping relation to structural and compound types in the usual way: Arrow types are contravariant in their domain types and covariant in their range types (i.e., $S \rightarrow T <: U \rightarrow V$ if and only if $U <: S$ and $T <: V$). One tuple type is a subtype of another if and only if they have the same number of elements, and each element of the first is a subtype of the corresponding element of the other (i.e., $(\overline{S}) <: (\overline{T})$ if and only if $|\overline{S}| = |\overline{T}|$ and $S_i <: T_i$ for all $1 \leq i \leq |\overline{S}|$). Intersection types are the most general subtypes of their element types, and union types are the most specific supertypes of their 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 understood to be empty, we write this judgment as simply $S <: T$. The subtype judgment can only be made on types $S, T \in \mathcal{T}$, so the judgment takes a class table $\mathcal{T}$ as an implicit parameter. The types S and T are said to be *equivalent*, written $S \equiv T$, when $S <: T$ and $T <: S$.

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

2.1 Values and Ilks

Types are intended to describe the values that might be produced by an expression or passed into a function or method. In Fortress, for example, there are three kinds of values: objects, functions, and tuples; every object belongs to at least one constructed type, every function belongs to at least one arrow type, and every tuple belongs to at least one tuple type. When we say that two types T and U have *the same extent*, we mean that for every value v, v belongs to T if and only if v belongs to U.

We place a requirement on values and on the type system that describes them: While a value may belong to more than one type, for every value v there is a unique type $\mathit{ilk}(v)$ (the *ilk* of the value) that is *representable in the type system*[2] and has the property that for every type T, if v belongs to T then $\mathit{ilk}(v) <: T$; moreover, $\mathit{ilk}(v) \neq \mathrm{BottomType}$. (This notion of ilk corresponds to what is sometimes called the "class" or "run-time type" of the value. 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, and we prefer it to the term "class," which is used in *The Java Language Specification* [9], 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.")

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

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

2.2 Overloaded Functions

A function declaration consists of a name, a sequence of type parameter declarations (enclosed in white square brackets), a type

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

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

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

For example, in the following function declaration:

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

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

A function declaration $f[\![\overline{X <: \{\overline{N}\}}]\!]\,S:T$ may be *instantiated* with type arguments $\overline{W}$ if $W_i <: [\overline{W/X}]N_{ij}$ for all i and j; we call $[\overline{W/X}]f\,S:T$ the *instantiation* of f with $\overline{W}$. When we do not care about $\overline{W}$, we just say that $f\,U:V$ is an *instance* of f.

We use the metavariable $\mathcal{D}$ to range over finite families of function declarations and write $\mathcal{D}(f)$ for the family in $\mathcal{D}$ that contains all declarations named f.

An instance $f\,U:V$ of a declaration f is *applicable* to a type W if and only if $W <: U$. A function declaration is *applicable* to a type if and only if at least one of its instances is.

For any two function declarations $f_1, f_2 \in \mathcal{D}(f)$, f_1 is *more specific* than f_2 (written $f_1 \preceq f_2$) if and only if for every type T such that f_1 is applicable to T, f_2 is also applicable to T.

We can now formulate the type safety properties required to hold for any set of overloaded function declarations as follows: We say that $\mathcal{D}(f)$ is *safe with respect to class table* $\mathcal{T}$ if and only if, for any type $W \in \mathcal{T}' \supseteq \mathcal{T}$ such that $W \neq \mathrm{BottomType}$[3], the following properties hold:

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

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

Static checking of these conditions guarantees that there are no ambiguous calls at run time due to a collection of overloaded function declarations. The progress condition ensures that we never get stuck resolving an application; every call to a function declaration is unambiguous. The preservation condition ensures that the static type of an application is a supertype of the ilk of each value produced by the application at run time, provided that the ilks for the argument values in function applications are always subtypes of the static types of the argument expressions (as in any language with a sound type system). As a monomorphic function declaration is a special case of a generic function declaration, where the sequence of the type parameters is empty, these conditions apply also for monomorphic function declarations.

[3] All functions are applicable to $\mathrm{BottomType}$, making the conditions quite trivial in that case. In any programming language that implements the present system, including Fortress, no values would ever have type $\mathrm{BottomType}$ and thus no function could ever apply to such a value at run time.

2.3 Type Inference

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

Aside from this caveat, our system for checking overloaded declarations is largely independent of how a specific type inference engine would choose instantiations[4]. Thus we do not discuss the specific features of a type inference system further in this paper.

3. Overloading Rules

Guaranteeing safety requires constraints on the sets of overloaded function declarations that may appear in a legal program [14, 15]. Some languages require program constructs that encapsulate all overloaded definitions; such constructs essentially guarantee safety "by construction". Examples in other languages include type classes [7, 20, 24] and multimethods [3, 14, 15]. We take a different approach: rather than introducing additional language facilities, we impose rules on the function declarations themselves. We intend these rules to be "minimal" in that they should be as unrestrictive as possible while preserving the ability to guarantee safety and be checked in a modular way. We took a similar appraoch to guarantee safety for overloaded monomorphic functions [1]. However, handling parametric polymorphism and implicit instantiation (i.e., type inference) requires more sophisticated type analysis. We achieve this analysis by introducing universal and existential quantification over the ground types defined by class tables.

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

3.1 Progress

Our proof strategy for satisfying the Progress condition makes use of the old idea from Castagna *et al.* [4] that for each name f the set of function declarations $\mathcal{D}(f)$ should form a meet semi-lattice under the *specificity* order defined in Section 2.2. If a declaration

[4] Type inference manifests itself as the choice of instantiation of type variables in existential and universal subtyping; specifically, $\overline{V}$ in the inference rules for subtyping in Figure 1. Mitchell [16] first showed how type inference interacts with polymorphic subtyping.

$$\boxed{\textbf{Universal subtyping:}\quad \Delta \vdash \sigma \le \sigma}$$

$$\frac{\begin{array}{cc}\Delta' = \Delta, \overline{Y <: \{\overline{N}\}} & \overline{Y} \cap FV(T) = \emptyset \\ \Delta' \vdash [\overline{V}/\overline{X}]T <: U & \overline{\Delta' \vdash V <: [\overline{V}/\overline{X}]\overline{M}}\end{array}}{\Delta \vdash \forall\llbracket\overline{X <: \{\overline{M}\}}\rrbracket T \le \forall\llbracket\overline{Y <: \{\overline{N}\}}\rrbracket U}$$

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

$$\boxed{\textbf{Existential subtyping:}\quad \Delta \vdash \delta \le \delta}$$

$$\frac{\begin{array}{cc}\Delta' = \Delta, \overline{X <: \{\overline{M}\}} & \overline{X} \cap FV(U) = \emptyset \\ \Delta' \vdash T <: [\overline{V}/\overline{Y}]U & \overline{\Delta' \vdash V <: [\overline{V}/\overline{Y}]\overline{N}}\end{array}}{\Delta \vdash \exists\llbracket\overline{X <: \{\overline{M}\}}\rrbracket T \le \exists\llbracket\overline{Y <: \{\overline{N}\}}\rrbracket U}$$

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

$$\boxed{\textbf{Applicability:}\quad \Delta \vdash \delta \ni T \quad f \ni T}$$

$$\frac{\Delta \vdash \exists\llbracket\rrbracket T \le \delta}{\Delta \vdash \delta \ni T}$$

$$\begin{array}{rcl}\delta \ni T & \stackrel{\text{def}}{=} & \emptyset \vdash \delta \ni T \\ f \ni T & \stackrel{\text{def}}{=} & \emptyset \vdash dom(f) \ni T\end{array}$$

$$\boxed{\textbf{Specificity:}\quad \Delta \vdash f \preceq f}$$

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

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

$$\boxed{\textbf{Existential meet:}\quad \delta_1 \wedge \delta_2}$$

$$\left(\exists\llbracket\overline{X <: \{\overline{M}\}}\rrbracket T\right) \wedge \left(\exists\llbracket\overline{Y <: \{\overline{N}\}}\rrbracket U\right) \quad \stackrel{\text{def}}{=} \quad \exists\llbracket\overline{X <: \{\overline{M}\}}, \overline{Y <: \{\overline{N}\}}\rrbracket(T \cap U) \quad \text{where} \quad \begin{cases}\overline{X} \cap \overline{Y} = \emptyset \\ \overline{X} \cap FV(U) = \emptyset \\ \overline{Y} \cap FV(T) = \emptyset\end{cases}$$

Figure 1. Subtyping on universal and existential types, applicability and specificity on generic function declarations, and meet of existential types

of name f is applicable to a type W, then the set $\mathcal{D}_W(f) \subseteq \mathcal{D}(f)$ of all declarations named f and applicable to W also forms a meet semi-lattice under specificity. As such, $\mathcal{D}_W(f)$ must have a least element.

We can phrase this condition more explicitly in the following two rules for a set of overloaded function declarations $\mathcal{D}$ and more-specific-than relation on these declarations by the subtyping relation induced by a class table $\mathcal{T}$:

No Duplicates Rule $\mathcal{D}(f)$ does not contain any two declarations that are equally specific (i.e., each declaration is more specific than the other).

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

To check the No Duplicates Rule, we need to determine if two declarations are equally specific. For this, we need a way to mechanically determine which instantiation of a generic declaration has an instance that is applicable to a type T. With a monomorphic declaration $f\,U\!:\!V$, we need only check that $T <: dom(f)$, where $dom(f)$ is the domain U of $U \to V$, the arrow type of f. We must then extend our notion of the "arrow type of f" for a generic function declaration f, but first we need to characterize the type of a generic function. In particular, we need a higher-level notion of universally quantified types separate from the language types T.

A *universal type* binds type parameter declarations over some type and can be instantiated by any types valid for those type parameters. We write $\forall\llbracket\overline{X <: \{\overline{M}\}}\rrbracket T$ to quantify each type variable X_i with bounds $\{\overline{M_i}\}$ over the type T, and we use the metavariable σ to range over universal types. Informally, the arrow type of a generic function declaration $f\llbracket\Delta\rrbracket\,U\!:\!V$ (written $arrow(f)$) is the universal type $\forall\llbracket\Delta\rrbracket U \to V$.

We know that f is applicable to T if and only if there exists some instantiation of f, $f\,U'\!:\!V'$ with $\overline{W}$, for which T is a subtype of the instantiated domain U'. This existential quantification over possible instantiations of the domain directly corresponds to an existentially quantified type as in [3]. An *existential type* $\exists\llbracket\overline{X <: \{\overline{M}\}}\rrbracket T$ also binds type parameter declarations over a type, but unlike a universal type, it cannot be instantiated; instead, it represents some hidden type instantiation $\overline{W}$ and the corresponding instantiated type $[\overline{W}/\overline{X}]T$. Therefore, we say that the domain of the above universal arrow type for f (again written $dom(f)$ as an abuse of notation) is the existential type $\exists\llbracket\Delta\rrbracket U$. We use the metavariable δ to range over existential types. Note that we abbreviate both the universal type $\forall\llbracket\rrbracket T$ and the existential type $\exists\llbracket\rrbracket T$ as simply T when the meaning is clear from context.

We define a pre-order on universal types corresponding to the subtype relation on types as originally given by Mitchell [16]: roughly, a universal type $\sigma_1 = \forall\llbracket\Delta_1\rrbracket T_1$ is a subtype of another universal type $\sigma_2 = \forall\llbracket\Delta_2\rrbracket T_2$ (written $\sigma_1 \le \sigma_2$) if T_1 can be instantiated to a subtype of T_2 in the environment Δ_2. We define a dual pre-order on existential types: an existential type $\delta_1 = \exists\llbracket\Delta_1\rrbracket T_1$ is a subtype of another existential type $\delta_2 = \exists\llbracket\Delta_2\rrbracket T_2$ (written $\delta_1 \le \delta_2$) if T_2 can be instantiated to a supertype of T_1 in the environment Δ_1. Figure 1 presents the syntactic judgments for these pre-orders and gives an expression $\delta_1 \le \delta_2$ for the meet of two existential types under $\le$. With subtyping on existential types, we can now formalize the notions of applicability and specificity for generic functions, also presented in Figure 1.

Lemma 1 $\delta_1 \wedge \delta_2$ is the meet of δ_1 and δ_2 under $\le$.

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

To check the Meet Rule, for every pair of declarations $f_1, f_2 \in \mathcal{D}(f)$, we must find a function declaration f_0 that is equivalent under specificity to the *computed meet* of f_1 and f_2, which is a declaration $f_\wedge$, not necessarily in $\mathcal{D}(f)$, such that $dom(f_\wedge) \equiv dom(f_1) \wedge dom(f_2)$[5]. More concretely, we find an f_0 such that:

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

Note that alpha renaming on the existential types' parameters might be necessary.

Note that subtyping is preserved under class table extension, so if $\mathcal{D}(f)$ satisfies the Meet Rule and the No Duplicates Rule with respect to the class table $\mathcal{T}$ then it satisfies them with respect to $\mathcal{T}'$ for any $\mathcal{T}' \supseteq \mathcal{T}$. Therefore, we can be certain that adding more types will not invalidate the Progress guarantee.

3.2 Preservation

We can ensure that the second well-formedness condition for overloading holds by checking the following rule for each function name f:

Return Type Rule For every $f_1, f_2 \in \mathcal{D}(f)$ with $f_1 \preceq f_2$, for every type W to which f_1 is applicable (and therefore f_2 is also applicable) and every instance $f_2'\, U_2 : V_2$ of f_2 that is applicable to W, there must exist an instance $f_1'\, U_1 : V_1$ of f_1 that is applicable to W and satisfies $V_1 <: V_2$.

To check the Return Type Rule, suppose $f_1 = f[\![\Delta_1]\!]\, S_1 : T_1$ $f_2 = f[\![\Delta_2]\!]\, S_2 : T_2$ such that $f_1 \preceq f_2$, W is a type to which f_1 is applicable, and the instance $f_2'\, S_2' : T_2'$ of f_2 is applicable to W. Let $f_\wedge = f[\![\Delta_1, \Delta_2]\!]\, S_1 \cap S_2 : T_1$. Then there is some instance $f_\wedge'\, U' : V'$ of $f_\wedge$ that is applicable to W with $V' = T_2'$. Therefore if we can just show that $arrow(f_1)$ is a subtype of $arrow(f_\wedge)$, which is true if and only if for every instantiation $f_\wedge'\, U' : V'$ of $f_\wedge$ that is applicable to W, there is an instantiation $f_1'\, S_1' : T_1'$ of f_1 with $U' <: S_1'$ and $T_1' <: V_2'$, then we know that f_1 and f_2 satisfy the Return Type Rule.

In summary, we know that f_1 and f_2 with $f_1 \preceq f_2$ satisfy the Return Type Rule whenever:

$$\forall[\![\Delta]\!] S \to T \;\leq\; \forall[\![\Delta, \Delta']\!](S \cap U) \to V$$

Once again, the fact that subtyping is preserved under class table extension lets us be sure that the property that $\mathcal{D}(f)$ is preserved under class table extension. Therefore the Return Type Rule and the rules from the last section are sufficient to ensure safety.

4. Examples and Problems

While the rules presented in section 3 guarantee the type safety of overloaded generic function declarations, they forbid many sets of overloadings that seem completely "natural". Unfortunately, these are not false negatives; many declarations that programmers would like to write are actually unsafe due to multiple inheritance.

For example, even though the following function declarations look like a valid overloading:

$$\begin{aligned} &\textit{simple}\ \text{String} : \text{String} \\ &\textit{simple}\ \mathbb{Z} : \mathbb{Z} \end{aligned}$$

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

they are not, because the Meet Rule is not satisfied. Moreover, in Fortress it is impossible to disambiguate the declarations by providing the meet because intersection types are not allowed in the Fortress function declaration syntax. In a language with single inheritance, we would be able to infer that these overloadings were safe due to the fact that a class can only have a single superclass. However, due to multiple inheritance, we cannot be sure that these types do not have a common subtype no matter what the programmer intends.

Now consider this less trivial set of overloaded functions:

$$\begin{aligned} &\textit{foo}[\![X <: \text{Any}]\!]\text{ArrayList}[\![X]\!] : \mathbb{Z} \\ &\textit{foo}[\![Y <: \mathbb{Z}]\!]\text{List}[\![Y]\!] : \mathbb{Z} \\ &\textit{foo}[\![W <: \mathbb{Z}]\!]\text{ArrayList}[\![W]\!] : \mathbb{Z} \end{aligned}$$

where $\text{ArrayList}[\![T]\!] <: \text{List}[\![T]\!]$ for all types T.[6] The first two declarations are incomparable under specificity—the first declaration applies to all instantiations of type constructor ArrayList, whereas the second declaration applies only to instantiations of type constructor List with subtypes of type $\mathbb{Z}$. The third definition, which is the obvious candidate to disambiguate the two, is not actually the meet; the domain of the meet candidate is the existential type:

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

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

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

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

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

This is not a problem with our definition of the meet: these declarations actually are unsafe. Consider the type:

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

Our meet candidate is not applicable to BadList, but the two other definitions of *foo* are, and so the Progress requirement is violated.

The following example does not work either.:

$$\begin{aligned} &\textit{tail}[\![X <: \text{Any}]\!]\text{List}[\![X]\!] : \text{List}[\![X]\!] \\ &\textit{tail}[\![Y <: \text{Any}]\!]\text{ArrayList}[\![Y]\!] : \text{ArrayList}[\![Y]\!] \end{aligned}$$

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

$$\begin{aligned} X <: \text{Any}, Y <: \text{Any} \vdash\ & \text{ArrayList}[\![V]\!] \to \text{ArrayList}[\![V]\!] \\ & <: \text{ArrayList}[\![Y]\!] \cap \text{List}[\![X]\!] \to \text{List}[\![X]\!] \end{aligned}$$

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

$$\textit{tail}\ \text{List}[\![\mathbb{Z}]\!] : \text{List}[\![\mathbb{Z}]\!]$$

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

$$\textit{tail}\ \text{ArrayList}[\![\text{String}]\!] : \text{ArrayList}[\![\text{String}]\!]$$

which violates the Preservation requirement.

[6] We use this standard declaration for ArrayList throughout.

5. Exclusion Relation

To enable overloaded functions of the sort described in the previous section to be defined, we introduce the *exclusion* relation $\diamondsuit$ on types: $S \diamondsuit T$ asserts that types S and T have no common subtypes other than $\mathrm{BottomType}$.

5.1 Exclusion Declaration

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

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

Omitting an `excludes` clause is equivalent to explicitly writing `excludes` { }, and omitting a `comprises` clause is equivalent to explicitly writing `comprises` { Any }.[7] For an application $C[\![\overline{T}]\!]$ of a declaration with the `excludes` and/or `comprises` clause above, we define the sets of instantiations of the types in these clauses analogously to $C[\![\overline{T}]\!]$.*extends*. That is,

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

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

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

The exclusion relation on constructed types can then be informally described as the combination of more precise sub-relations on those types that each corresponds to a certain reason, or proof, of exclusion:

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

Exclusion between constructed types is informally defined as the union of the sub-relations $\rhd_{\mathrm{e}}$, $\rhd_{\mathrm{c}}$, and $\rhd_{\mathrm{o}}$. (We shall introduce another sub-relation in the next subsection.)

The exclusion relation induced by a class table on all its types is then the symmetric closure of the relation derived from the constructed types as described above, extended to structural and compound types as follows: Every arrow type excludes every non-arrow type. Every tuple type excludes every non-tuple type and also every tuple type with a different number of element types. An intersection type excludes any type excluded by *any* of its constituent types, while a union type excludes any type excluded by *all* of its constituent types. BottomType excludes every type (including itself—it is the only type that excludes itself), and Any does not exclude any type other than BottomType. (Later in Section 5.2 we shall define the exclusion relation formally.) We augment our notion of a well-formed class table to require that the subtyping and exclusion relations it induces "respect" each other. That is, for all constructed types M and N other than $\mathrm{BottomType}$,

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

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

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

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

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

However, as we discussed in the previous section, checking well-formedness of the declarations $\mathcal{D}(tail)$ is trickier. Certainly neither List nor ArrayList declares an exclusion of the other, so seemingly the exclusion relation does not help us check these declarations.

A fundamental problem with $\mathcal{D}(foo)$ is that a type such as BadList might be a subtype of multiple instantiations of type constructor List (in this case, a subtype of $\mathrm{List}[\![\mathbb{Z}]\!]$ directly and of $\mathrm{List}[\![\mathrm{String}]\!]$ via $\mathrm{ArrayList}[\![\mathrm{String}]\!]$). A type hierarchy in which some type extends multiple instantiations of the same type constructor is said to exhibit *multiple instantiation inheritance* [12].

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

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

If a type M is a subtype of an application $C[\![\overline{T}]\!]$ of type constructor C, then M excludes all other applications $C[\![\overline{U}]\!]$ of C. We augment our exclusion relation on constructed types with this final sub-relation, which we write as $M \rhd_{\mathrm{p}} C[\![\overline{U}]\!]$. This restriction is easy to statically enforce, and it has been shown in practice that it is not an onerous restriction. Indeed, it is already in the Java™ programming language, and for good reason: Kennedy and Pierce showed that multiple instantiation inheritance is one of three conditions that lead to the undecidability of nominal subtyping with variance [12].

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

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

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

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

5.2 Exclusion with Constraints

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

Our grammar for type constraints is defined as follows:

$$\begin{array}{lcll} \mathcal{C} & ::= & X <: S & \text{primitive constraint} \\ & | & S <: X & \text{primitive constraint} \\ & | & \mathcal{C} \wedge \mathcal{C} & \text{conjunction constraint} \\ & | & \mathcal{C} \vee \mathcal{C} & \text{disjunction constraint} \\ & | & \text{false} & \text{never satisfied} \\ & | & \text{true} & \text{always satisfied} \end{array}$$

A primitive constraint of the form $X <: S$ specifies that a type variable X is a subtype of S, and likewise for $S <: X$. A conjunction constraint $\mathcal{C}_1 \wedge \mathcal{C}_2$ is satisfied exactly when both $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfied, and a disjunction constraint $\mathcal{C}_1 \vee \mathcal{C}_2$ is satisfied exactly when one or both of $\mathcal{C}_1$ and $\mathcal{C}_2$ are satisfied. The constraint false is never satisfied, and the constraint true is always satisfied. We introduce the following syntactic judgments to generate these constraints:

$$\Delta \vdash S <: T \,|\, \mathcal{C} \quad \Delta \vdash S \equiv T \,|\, \mathcal{C} \quad \Delta \vdash S \lozenge T \,|\, \mathcal{C}$$

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

We do not list here the full semantics of constraint generation for subtyping. Instead we refer the reader to the recent work by Smith and Cartwright [22] on which our system was based. Smith and Cartwright provide a sound and complete algorithm for generating constraints from the subtyping relation and an algorithm for normalizing constraints to simplify, for example, those involving contradictions or redundancies. We assume that constraints are implicitly simplified in this manner. The soundness of constraint generation entails that the predicate is a logical consequence of the constraint; the completeness of generation entails the opposite.

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

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

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

To define the constraint generation judgment for exclusion, we need to know the constraint under which two types are not equivalent and one is not a subtype of the other. For this, we introduce new forms of primitive constraints to our grammar

$$\begin{array}{lcll} \mathcal{C} & ::= & \ldots & \\ & | & X \not<: S & \text{primitive constraint} \\ & | & S \not<: X & \text{primitive constraint} \end{array}$$

which we call *negatives*; we call the original primitive constraints *positives*. With an even richer language of constraints, we can now define a new judgment for "not a subtype" by the rule

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

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

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

These judgements are sound because constraint generation for $<:$ is complete (and complete because $<:$ is sound).
With these additional constraints and judgments, we can now define constraint generation for exclusion. Our algorithm—which formalizes our original definition of exclusion from Section 5.1—is presented as inference rules in Figure 2. As before, exclusion on constructed types is satisfied when any of the sub-relations from the previous section is satisfied. Each sub-relation checks the conditions described before by recursively generating and propagating constraints. Each sub-relation except $\triangleright_c$ depends only on other constraint generation judgments, meaning the algorithmic checking terminates. The $C[\![\overline{T}]\!] \triangleright_c D[\![\overline{U}]\!]$ predicate recursively checks exclusion on the comprised types of $C[\![\overline{T}]\!]$, but due to the acyclic class table, this process will also terminate. The algorithm is complete and sound if $<:$ is.

For subtyping with constraints, we essentially preserve the semantics of [22], but that system lacks our notion of exclusion. Since we certainly need our subtyping to take advantage of exclusion, we must add an additional rule to the judgment:

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

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

We can recover the usual subtyping judgment and define the exclusion judgment as the following

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

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

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

Symmetry

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

Structural rules

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

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

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

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

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

$$\frac{C \neq \mathrm{Any} \qquad |\overline{T}| \neq 1}{\Delta \vdash C\llbracket \overline{S} \rrbracket \lozenge (\overline{T}) \mid \text{true}}$$

$$\frac{C \neq \mathrm{Any}}{\Delta \vdash C\llbracket \overline{S} \rrbracket \lozenge (T \to U) \mid \text{true}}$$

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

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

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

Type variables

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

Constructed types

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

$$\frac{\Delta \vdash C\llbracket \overline{S} \rrbracket \triangleright_x D\llbracket \overline{T} \rrbracket \mid \mathcal{C} \qquad \Delta \vdash D\llbracket \overline{T} \rrbracket \triangleright_x C\llbracket \overline{S} \rrbracket \mid \mathcal{C}'}{\Delta \vdash C\llbracket \overline{S} \rrbracket \lozenge_x D\llbracket \overline{T} \rrbracket \mid \mathcal{C} \vee \mathcal{C}'}$$

where x may be ‘e’, ‘o’ or ‘c’.

$$\frac{C = D}{\Delta \vdash C\llbracket \overline{S} \rrbracket \triangleright_x D\llbracket \overline{T} \rrbracket \mid \text{false}}$$

where x may be ‘e’, ‘o’ or ‘c’.

$$\frac{\texttt{object}\ C \qquad C \neq D \qquad \Delta \vdash C\llbracket \overline{S} \rrbracket \not<: D\llbracket \overline{T} \rrbracket \mid \mathcal{C}}{\Delta \vdash C\llbracket \overline{S} \rrbracket \triangleright_{\mathrm{o}} D\llbracket \overline{T} \rrbracket \mid \mathcal{C}}$$

$$\frac{\neg(\texttt{object}\ C)}{\Delta \vdash C\llbracket \overline{S} \rrbracket \triangleright_{\mathrm{o}} D\llbracket \overline{T} \rrbracket \mid \text{false}}$$

$$\frac{C \neq D \qquad \forall U \in \textit{excludes}^*(C\llbracket \overline{S} \rrbracket).\ \Delta \vdash D\llbracket \overline{T} \rrbracket <: U \mid \mathcal{C}_U}{\Delta \vdash C\llbracket \overline{S} \rrbracket \triangleright_{\mathrm{e}} D\llbracket \overline{T} \rrbracket \mid \bigvee \mathcal{C}_U}$$

where $\textit{excludes}^*(M) = \bigcup_{N \in \textit{ancestors}(M)} N.\textit{excludes}$

$$\frac{C \neq D \qquad \forall U \in C\llbracket \overline{S} \rrbracket.\textit{comprises}.\ \Delta \vdash U \lozenge D\llbracket \overline{T} \rrbracket \mid \mathcal{C}_U \qquad \Delta \vdash C\llbracket \overline{S} \rrbracket \not<: D\llbracket \overline{T} \rrbracket \mid \mathcal{C}}{\Delta \vdash C\llbracket \overline{S} \rrbracket \triangleright_{\mathrm{c}} D\llbracket \overline{T} \rrbracket \mid \mathcal{C} \wedge \bigwedge \mathcal{C}_U}$$

$$\frac{C \text{ does not have a } \texttt{comprises} \text{ clause}}{\Delta \vdash C\llbracket \overline{S} \rrbracket \triangleright_{\mathrm{c}} D\llbracket \overline{T} \rrbracket \mid \text{false}}$$

$$\frac{\forall U \in \textit{ancestors}(C\llbracket \overline{S} \rrbracket), V \in \textit{ancestors}(D\llbracket \overline{T} \rrbracket).\ \Delta \vdash U \blacklozenge_{\mathrm{p}} V \mid \mathcal{C}_{U,V}}{\Delta \vdash C\llbracket \overline{S} \rrbracket \lozenge_{\mathrm{p}} D\llbracket \overline{T} \rrbracket \mid \bigvee \mathcal{C}_{U,V}}$$

$$\frac{C = D \qquad \Delta \vdash \overline{S} \not\equiv \overline{T} \mid \mathcal{C}}{\Delta \vdash C\llbracket \overline{S} \rrbracket \blacklozenge_{\mathrm{p}} D\llbracket \overline{T} \rrbracket \mid \mathcal{C}}$$

$$\frac{C \neq D}{\Delta \vdash C\llbracket \overline{S} \rrbracket \blacklozenge_{\mathrm{p}} D\llbracket \overline{T} \rrbracket \mid \text{false}}$$

Figure 2. Generating constraints for exclusion

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

Solving constraints needs to be sound but not necessarily complete. Here is a simple algorithm which is sound but not complete:

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

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

5.3 Tying the Knot

We now use the non-equivalence judgment to make rigorous our intuition about how polymorphic exclusion affects the mechanical verification of the overloading rules. We define a reduction judgment on existential types in Figure 3 which reduces δ to δ' such that $T \ni \delta$ if and only if $T \ni \delta$. Furthermore we have that $\delta' \leq \delta$. Then we will show how augmenting the $\leq$ relation with the rule

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

lets us prove that more sets of overloaded functions satisfy the overloading rules. Reducing an existential type in this fashion involves the same kind of type analysis required in other languages for type checking generalized algebraic data types [19, 21].

The reduction judgment $\vdash \delta \xrightarrow{\equiv} \delta',\ \phi$ says that the existential type $\delta = \exists[\![\Delta]\!]T$ reduces to $\delta' = \exists[\![\Delta']\!]T'$ with the substitution ϕ (when the substitution is unnecessary we omit it) under the assumption that T is not equivalent to $\mathrm{BottomType}$; or, if T is equivalent to $\mathrm{BottomType}$, the reduced existential δ' is $\mathrm{BottomType}$. As an example, consider the following instance of reduction

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

$$\xrightarrow{\equiv}$$

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

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

Once we have augmented the subtyping relation with existential reduction, we can finally check that the declarations $\mathcal{D}(foo)$ from Section 4 satisfy the Meet Rule.

A similar analysis shows that if an instance of a universal arrow has the domain $\mathrm{BottomType}$, then it is irrelevant for the purposes of guaranteeing Progress. Therefore we can use our reduction rule for existential types to aid in the verification of the Return Type Rule by augmenting the subtype rules for universal arrows.

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

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

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

$$\boxed{\textbf{Bounds substitution:}\quad \phi[\Delta] = \Delta}$$

$$\frac{\begin{array}{c}\Delta = \overline{X <: \{\overline{M}\}} \qquad \phi(\overline{X}) = \overline{Y} \sqcup \overline{T} \qquad \overline{N} = \phi^{-1}[\overline{Y}, \Delta] \\ \forall i.\quad \overline{Y <: \{\overline{N}\}} \vdash \phi(X_i) <: \{\overline{\phi(M_i)}\} \mid \mathcal{C}_i \qquad \mathcal{C}_i = \mathrm{true}\end{array}}{\phi[\Delta] = \overline{Y <: \{\overline{N}\}}}$$

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

$$\frac{\overline{T} = \{\phi(\Delta(X))\ :\ X \in \Delta \text{ and } \phi(X) = Y\}}{\phi^{-1}[Y, \Delta] = conjuncts(\bigcap \overline{T})}$$

Figure 3. Reduction of existential types

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

It is easy to see from the previous existential reduction instance that this subtype judgment can be proven, thus allowing us to verify the Return Type Rule for the function $\mathcal{D}(tail)$ from Section 4.

6. Related Work

Overloading and dynamic dispatch Primarily, our system strictly extends our previous effort [1] with parametric polymorphism; all previous properties and results remain intact. The inclusion of parametric functions and types represents a shift in the research literature on overloading and multiple dynamic dispatch.

Millstein and Chambers [14, 15] devised the language Dubious to study overloaded functions with symmetric multiple dynamic dispatch (*multimethods*), and with Clifton and Leavens they developed MultiJava [5], an extension of Java with Dubious' semantics for multimethods. In [13], Lee and Chambers presented F(EML), a language with classes, symmetric multiple dispatch, and parameterized modules, but without parametricity at the level of functions or types. None of these systems support polymorphic functions or types. F(EML)'s parameterized modules (*functors*) constitute a form of parametricity but they cannot be implicitly applied; the functions defined therein cannot be *overloaded* with those defined in other functors. For a more detailed comparison of modularity and dispatch between our system and these, we refer to the related work section of our previous paper [1].

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

Type classes Wadler and Blott [24] introduced the *type class* construct as a means to specify and implement overloaded functions like equality and arithmetic operators in Haskell. Other authors have translated type classes to languages besides Haskell [7, 20, 25]. 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:

$$\begin{aligned} &\mathit{println}()\colon () = \mathit{println}(\text{``''}) \\ &\mathit{println}(s\colon \mathrm{String})\colon () = \ldots \end{aligned}$$

or functions lacking uniform variance in the domain and range[8] like the following:

$$\begin{aligned} &\mathit{bar}(x\colon \mathbb{Z})\colon \mathrm{Boolean} = (x = 0) \\ &\mathit{bar}(x\colon \mathrm{Boolean})\colon \mathbb{Z} = \texttt{if}\ x\ \texttt{then}\ 1\ \texttt{else}\ 2\ \texttt{end} \\ &\mathit{bar}(x\colon \mathrm{String})\colon \mathrm{String} = x \end{aligned}$$

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. In Appendix A, we present a further discussion of how our overloading resolution differs from that of Haskell and how our system might translate to that language, thereby addressing an existing inconsistency in modern type class extensions.

A broader interpretation of Wadler and Blott's [24] 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.

7. 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 programming language's compiler [2].

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

Variance annotations have proven to be a convenient and expressive addition to languages based on nominal subtyping [3, 12, 17]. They add additional complexity to polymorphic exclusion checking, so we leave them to future work.

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

[2] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification Version 1.0, March 2008. URL `http://labs.oracle.com/projects/plrg/Publications/fortress.1.0.pdf`.

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

[4] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *SIGPLAN Lisp Pointers*, V(1):182–192, 1992.

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

[6] D. Doel and S. Peyton Jones. Overlapping instances + existentials = incoherent instances, February 2010. URL `http://www.haskell.org/pipermail/glasgow-haskell-users/2010-February/018417.html`. Glasgow Haskell Users mailing list.

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

[8] K.-F. Faxén. A static semantics for Haskell. *J. Funct. Program.*, 12 (5):295–357, 2002.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[10] C. Hall, K. Hammond, S. Peyton Jones, and P. Wadler. Type classes in Haskell. *ACM Trans. Program. Lang. Syst.*, 18:241–256, 1996.

[11] A. Hejlsberg, S. Wiltamuth, and P. Golde. *C# Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[12] A. J. Kennedy and B. C. Pierce. On decidability of nominal subtyping with variance, September 2006. FOOL-WOOD '07.

[13] K. Lee and C. Chambers. Parameterized modules for classes and extensible functions. In *ECOOP '06 : Proceedings of the 20th European Conference on Object-Oriented Programming*. Springer-Verlag, 2006.

[14] T. Millstein and C. Chambers. Modular statically typed multimethods. In *Information and Computation*, pages 279–303. Springer-Verlag, 2002.

[15] T. D. Millstein. *Reconciling software extensibility with modular program reasoning*. PhD thesis, University of Washington, 2003.

[16] J. C. Mitchell. Polymorphic type inference and containment. *Inf. Comput.*, 76(2-3):211–249, 1988.

[17] M. Odersky. *The Scala Language Specification, Version 2.7.* EPFL Lausanne, Switzerland, 2009. URL `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.

[18] S. Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

[19] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 341–352, New York, NY, USA, 2009. ACM.

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

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

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

[23] The GHC Team. The Glorious Glasgow Haskell Compilation System User's Guide, 2010. URL `http://www.haskell.org/ghc/docs/6.12.2/users_guide.pdf`. Version 6.12.2.

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

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

A. Application to Haskell

The Haskell 98 definition [18] prevents two instances of the same class from containing the same type. The prevalent Glasgow Haskell Compiler (GHC), however, allows such *overlapping instances* among other commonplace type class features [23, §7.6]. With this feature, one can write default and specialized implementations of class methods. For example, the following code for counting things:

```
class Things a where
    things :: a -> Int

instance Things a where      -- (1)
    things _ = 1
instance Things [a] where    -- (2)
    things xs = length xs
instance Things String where -- (3)
    things _ = 1

f :: [b] -> Int
f x = things x
```

includes overlapping instances of `Things`. The instances (2) and (3) overlap because `String` is a synonym for `[Char]`, and clearly both overlap with the fully general instance (1). Instance (3) is included to guarantee that `things s = 1` for any string (i.e. list of characters) `s`.

One might expect `(f "foo")` to evaluate to `1`, but instead it evaluates to `3`. When GHC checks the call of `things` inside the body of `f`, it must find evidence for the constraint `(Things [b])` generated by applying `things` to an expression of type `[b]`. If `f`'s signature denoted the constraint `(Things [b])` as an assumption, such evidence clearly exists; without that assumption, as is the case here, the evidence can only be provided by the declared instances. Thus, GHC resolves this call with the most fitting choice (2), despite the possibility that at runtime `b` could be instantiated with `Char`, making the more specific instance (3) an even better choice.

Such overlapping instances are called *incoherent*, and without enabling *incoherent instances* feature, these declarations would be rejected[9]. However, as demonstrated recently in [6], the same behavior occurs with the popular *generalized algebraic data types* (GADTs) feature enabled. Consider packaging a list of countable things into a GADT:

```
data SomeThings where
  MkST :: Things a => [a] -> SomeThings

g :: SomeThings -> Int
g (MkST ys) = things ys
```

As before, `g (MkST "foo")` evaluates to `3` instead of `1`, but there is no way to evaluate to `1` in this case. In the previous case, one could qualify the signature of `f` with the assumption `(Things [b])` to avoid necessarily selecting the general instance on lists, (2), but here no such qualification on `g` is even syntactically possible. Moreover, the following, seemingly equivalent expressions evaluate to `3` and `1` respectively:

```
h3 = case (MkST "foo") of (MkST ys) -> things ys
h1 = things "foo"
```

At the heart of this inconsistent behavior lies the static yet staggered nature of instance resolution in Haskell: *static* because instance resolution is determined while type checking [8, 10], and *staggered* because an instance is either resolved on a particular class method call or passed in from elsewhere in the call graph. If a more specific instance could override the statically determined instance at run time, such inconsistency would never occur. Instead of program behavior being determined by the static, possibly staggered evidence, it would be determined precisely by all the evidence available at run time. (The static evidence then serves as a guarantee that some exists; whether that evidence is actually used depends on the presence of *subevidence* during execution.) For example, when the body of `f` is executed with `b = Char`, the execution could hot swap the more specific instance (3) in place of the statically determined (2). Likewise in the case of `g`, if the GADT contains a single string (i.e. list of characters) like `"foo"`, the execution could override the statically determined (2) with (3). Such behavior could then be described as a kind of dynamic dispatch.

With this notion of subevidence and dynamic dispatch in Haskell, how can we be sure that there will always be a *unique* best choice of instance at run time? Would choosing a more specific instance at run time respect type safety? Such concerns are exactly those addressed by our system.

[9] GHC language features required for this example: `-XFlexibleInstances`, `-XTypeSynonymInstances`, `-XOverlappingInstances`, `-XIncoherentInstances`