Towards Practical Runtime Type Instantiation

Karl Naden
Carnegie Mellon University
kbn@cs.cmu.edu

Abstract

Symmetric multiple dispatch, generic functions, and variant type parameters are powerful language features that have been shown to aid in modular and extensible library design. However, when symmetric dispatch is applied to generic functions, type parameters may have to be instantiated as a part of dispatch. Adding variant generics increases the complexity of type instantiation, potentially making it prohibitively expensive. We present a syntactic restriction on generic functions and an algorithm designed and implemented for the Fortress programming language that simplifies the computation required at runtime when these features are combined.

1. Introduction

Symmetric multiple dispatch brings with it several benefits for writing modular and efficient code. It provides a partial solution to both the binary method problem and the expression problem and also ensures that the implementation with the most knowledge of the representations of the inputs is executed [2]. In exchange, the runtime must do extra work at each call site to determine the most specific function declaration that applies to the runtime arguments. Part of checking a generic function for applicability is finding a valid instantiation of its type parameters. This potentially intensive operation involves the evaluation of bounds on the type parameters and is made more difficult by variant generic types because they can increase the number of valid instantiations.

Existing work avoids the need to instantiate type bounds at runtime either by not including generics [2] or by requiring that the generic signature of each function in an overload set is the same which allows the instantiation to be generated at compile time [1]. However, designers of the Fortress programming language [3] found it useful while writing their standard library to define both monomorphic and polymorphic versions of functions [4]. In order to support this, we need to develop strategies to overcome the challenges of runtime type instantiation.

In this preliminary work, we show that restricting the scope of type parameters so that they may only appear after their bounds are declared serves to keep the runtime computation for dispatch to generic functions reasonable. We introduce an algorithm that requires only a single pass over the type parameters of a generic function in a subset of Fortress with co- and invariant generics.

2. Dispatch Problem Statement

In contrast with traditional dynamic dispatch, the runtime of a language with multiple dispatch cannot necessarily use any static information about the call site provided by the typechecker. It must check if there exists a more specific function declaration than the one statically chosen that has the same name and is applicable to the runtime types of the provided arguments. The process for determining when a function declaration is more specific than another is laid out in [4]. A fully instantiated function declaration is *applicable* if the runtime types of the arguments are subtypes of the declared parameter types. To show applicability for a generic function we need to find an instantiation that witnesses the applicability and the type safety of the function call so that it can be used by the code.

Formally, given

1. a function signature $f[\![\overline{X <: \tau_X}]\!](\overline{y : \tau_y}) : \tau_\mathrm{r}$,
2. the runtime types $\overline{T_y}$ of the runtime arguments provided to the function call, and
3. the return type T_r of the statically most specific applicable function signature,

f is applicable if we can provide an instantiation $\overline{T_X}$ of the type parameters of f such that for each type parameter X we have $T_X <: [\overline{T_X/X}]\tau_X$, for each runtime parameter y we have $T_y <: [\overline{T_X/X}]\tau_y$, and $[\overline{T_X/X}]\tau_\mathrm{r} <: T_\mathrm{r}$.

2.1 Language

In the above definition, the type parameters of f are declared in between oxford brackets ($[\![]\!]$). Each type parameter X can have multiple upper bounds τ_X. τ denotes possibly generic types, of the form X or $\texttt{Foo}[\![\overline{\tau}]\!]$, where `Foo` is a declared generic nominal type. A concrete type T is any fully instantiated nominal type. Nominal types are declared with type parameters, each of which is given a *variance* that defines how different instantiations of the type are related by subtyping. Concretely, given a generic nominal type $\texttt{Bar}[\![X]\!]$ where X is `covariant` we can say that $\texttt{Bar}[\![T_1]\!] <: \texttt{Bar}[\![T_2]\!]$ if and only if $T_1 <: T_2$. If X is `invariant`, then we must have $T_1 = T_2$.

In order to guarantee termination of our algorithm, we prevent type parameters from appearing in bounds prior to their declaration, including their own.[1] As we will see, this is required to prevent (infinite) iteration during bound generation.

3. Solution

To solve this problem, we generate all of the concrete upper and lower bounds on the type parameters from the information provided and then check that the bounds are consistent. That is, for each type parameter we check that the join of all of the lower bounds is a

[1] This restriction can be relaxed to allow the declared parameter to appear in its own bound in the case of *selftypes*.

subtype of the meet of all of the upper bounds. Since this could leave us with multiple valid instantiations we choose the lower bound of each type parameter because that will make the return type as specific as possible.

3.1 Bound Generation

Our first task is to generate all of the concrete upper and lower bounds for each type parameter. Upper bounds on type parameters that are declared as concrete types require no extra processing, but other bounds require more work.

3.1.1 Parameter Bounds

For an instantiation to be valid, the instantiated parameter types must be supertypes of the runtime types of the arguments. For each parameter y, we use the subtype inequality $T_y <: \tau_y$ to either prove that no instantiation exists or to generate bounds on any type variables in τ_y that ensure the instantiated function is applicable to this parameter. We do this using a recursive process called *structure matching*. Structure matching works because the type on the left is a concrete runtime type which we can query. For instance, if $\tau_y = \texttt{Baz}[\![X]\!]$, we can determine what instantiation T'_y of `Baz` is a supertype of T_y (as in [5] we require that this query return the unique minimal answer) and use the answer to generate bounds on X (if no T'_y exists then we know no instantiation will make the function declaration applicable). If the type parameter of `Baz` is `covariant`, we add T'_y as a lower bound; if it is `invariant`, we add T'_y as both a lower and an upper bound. As there can be arbitrary nesting, we may have to repeat this process a finite number of times in order to reach the type variable base case.

3.1.2 Return Type Bounds

In order to be type safe, the return type of a valid instantiation must be a subtype of the return type for the function call guaranteed by the typechecker. Unlike parameter bounds, these bounds do not rely on runtime information. Thus, we can generate additional concrete bounds on type parameters for more specific function declarations at each call site.

3.1.3 Declared Generic Upper Bounds

Subtype inequalities with generics on both sides require more care to ensure that we don't get into fixed-point iteration, which could be unbounded or require a prohibitive number of steps. We can avoid iteration by computing bounds in a smart order. For this purpose we define *bound dependence* between the bounds of type parameters. This relationship signifies the order that the bounds computation must be completed to avoid iteration. Specifically, if the lower bound of X is dependent on the lower bound of X', then all of the lower bounds of X' must be computed before we know we have generated all of the lower bounds of X.

Type parameter upper bounds. If we have $X <: Z$, then all lower bounds of X are also lower bonds of Z. Thus we say that the lower bound of Z is dependent on the lower bound of X. Similarly, all upper bounds of Z are also upper bounds of X, so the upper bound of X is dependent on the upper bound of Z.

Generic nominal upper bounds. If a nominal generic type appears in the upper bound as in $X <: \tau_X$, then we have a situation similar to the parameter bound question because we have a generic type as the super type. Unlike the parameter case, the subtype is a type variable and not a concrete type. Therefore, we cannot immediately use structure matching because we don't have a concrete type to query. However, the type we want to query is the lower bound of X because that is what we ultimately need to guarantee is a subtype of the upper bound. Therefore, any bounds we would generate by processing this generic bound are dependent on the lower bound of X. Once we have all of the lower bounds of X, we can compute the join of its lower bounds and use the join to generate bounds with structure matching.

3.2 Algorithm

Our algorithm is as follows:

1. Compute parameter bounds via structure matching.
2. In reverse declaration order, compute the lower bounds of each type parameter and then use structure matching to generate concrete bounds from its declared upper bounds.
3. In declaration order, compute the upper bounds of the type parameters and check that the computed lower bound is a subtype of the computed upper bound.
4. Choose the lower bounds of each type parameter as the instantiation.

We claim that this algorithm is guaranteed to terminate in a single pass. The only place iteration could occur is in step 2: if a new lower bound is added to an already processed type parameter, X, we need to update its computed lower bound and structure match again. However, our restriction on the scoping of type parameters ensures that the lower bounds of X can only be dependent on the lower bounds of type parameters that are declared after X. So, processing X and type parameters declared before X will not generate new lower bounds for X and so we will not need to iterate.

4. Conclusions and Future Work

We conclude that simply restricting the scope of type parameters in bounds is enough to guarantee a single pass algorithm. While we have not performed a complete evaluation of the impact of this restriction, we believe it is reasonable because all functions in the Fortress standard library conform to it. We have implemented the algorithm in the open source Fortress runtime.[2]

This is only the first step towards a reasonable implementation that maintains the power of generics. A next step would be to add contravariant generics, which will require an equivalent to structural matching when the subtype is generic. This algorithm also depends on the efficient computation of meets, joins, and subtypes over the type lattice at runtime. Finally, this algorithm may need to iterate over multiple signatures at a given call site. We are actively investigating representations and algorithms for these problems.

Acknowledgments

We thank David Chase, Justin Hilburn, Guy Steele, Victor Luchangco and the rest of the Fortress team for many helpful discussions.

References

[1] François Bourdoncle and Stephan Merz. Type checking higher-order polymorphic multi-methods. In *Proceedings of POPL '97*, pages 302–315, New York, NY, USA, 1997.

[2] Curtis Clifton et al. MultiJava: Design rationale, compiler implementation, and applications. *TOPLAS*, 28(3):517–575, 2006.

[3] Eric Allen et al. The Fortress Language Specification Version 1.0, March 2008.

[4] Eric Allen et al. Type checking modular multiple dispatch with parametric polymorphism and multiple inheritance. In *Proceedings of OOPSLA '11*, New York, NY, USA, 2011.

[5] Ross Tate et al. Taming wildcards in Java's type system. *SIGPLAN Not.*, 46:614–627, June 2011.

[2] http://projectfortress.java.net/