Parameterized Types for GOO

by

James Knight

S.B. in Electrical Engineering and Computer Science from the
Massachusetts Institute of Technology (2002)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Aug 2002

Author . . . .
Department of Electrical Engineering and Computer Science
Aug 28, 2002

Certified by . . . .
Jonathan R. Bachrach
Research Scientist
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Parameterized Types for GOO

by

James Knight

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 28, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

An often-useful addition to many programming languages is a generic type or parameterized type system. I have added support for a parameterized type system to the programming language "GOO". Besides the standard functionality, I have allowed the programmer to express some unique relationships between parameterized types that are generally inexpressible in a standard type system, in order to make the system applicable to a wider variety of situations. I have also added a unification system to the method dispatch in GOO in order to allow the relationship between the types of multiple arguments in a method signature to be expressed.

Thesis Supervisor: Jonathan R. Bachrach
Title: Research Scientist

Acknowledgments

I would like to thank the Dynamic Languages group at the MIT Artificial Intelligence Laboratory for funding this research. I would especially like to thank Jonathan Bachrach for supervising the project and providing me with an interesting programming language with which to work. I would also like to thank Gregory Sullivan and Neel Krishnaswami for giving a great number of useful suggestions.

Contents

List of Figures

Chapter 1

Introduction

Parameterized types, also known as generic types, are a fairly common extension to standard class-based type systems. In a class-based type system, one of the major deficiencies is the inability to precisely describe container types, such as an "Array of integers". Unless one manually creates distinct classes of Array for holding each type of object, it is impossible for the compiler to know the type of object returned when retrieving an object from the array. This lack of information is an issue for both the compiler, since it does not have as much information with which to generate optimizations, and for the programmer, as no compile-time guarantees about type safety are possible. By adding the ability to parameterize classes with additional arguments, it is possible to specify more precise types, without substantially bloating the class hierarchy.

GOO[Bac02] is a new language being developed by Jonathan Bachrach. GOO's design is influenced heavily influenced both by Dylan and CLOS. It supports multiple dispatch, with all parameters participating in the dispatch. GOO also has a fairly rich type system, including singletons, unions, and product types.

My work has been to add a parameterized type system to GOO. In my implementation, I have tried to keep to the ideals of GOO—that is, to be as dynamic and extensible as possible, while keeping the interface simple and concise. It is based on the substantial amount of prior work in this area, but goes beyond that already done in its flexibility and usefulness.

Because of the rich multimethod dispatching available in GOO, adding parameterized types to the language also necessitates adding type unification to the dispatch system in order for the parameterized types to actually be useful. The unification system introduces free type variables which allow type information to be shared amongst different arguments in the signature, so that the type of one argument can be related to the type of another. This unification system comprises half of the work.

With the GOO implementation, however, more complex relationships can be expressed, e.g. that an “Array of integer” is a subtype of “Array of (subtype-of number)”. These relationships are important in many real scenarios. This addition to the type system comprises the other half of the work.

Chapter 2

Related Work

There are a few varieties of parameterized types in use today. The most well-known form of parameterization is the primitive array type in C and Java. In these languages, you can create an array containing objects of a specific type using special square bracket syntax. However, it is not possible to create your own parameterized types. In Java, it feels a lot more limiting, as there is a strange disconnect between the primitive array type which can be parameterized, but cannot be resized, and the "Vector" and "ArrayList" types which cannot be parameterized, but can be resized.

Dylan[Sha96] has a similar concept called "limited types" which allows parameterization of the built-in collection classes. Unlike Java, the limited type system covers all the built-in collections types. However, it cannot do more than that. It is simply a special case in the compiler which cannot be extended to cover user-created types.

C++[Str91] has a very powerful templating system. It allows parameterizing on types and on primitive values, such as true, false, integers, and characters. However, after going through a sophisticated compile-time system for determining template parameters (it can do integer arithmetic and also has a type inference system for deducing the function template parameters when they are unspecified), all it really ends up doing is a simple textual substitution of the parameters into the class definition. Each substitution creates a new actual class in the compiled output. This has two advantages: there is no special runtime support needed, and the code can be optimized for every set of parameters. However, there are also disadvantages. While good for optimization, compiling a separate copy of the code for every set of parameters mentioned in the code can result in significant bloat, especially if there are many parameters. The other major disadvantage is that C++ does not provide any way to relate two instantiations of a template in the type-hierarchy.

ML[Pau91] has a form of parameterization based on modules. Essentially, by filling in missing bindings in a module, you can create a new module with customized behavior. This is very similar to C++'s templating system, and is similarly lacking in expressiveness in the type hierarchy. ML also has a concept called parametric types, which is the only way you can have methods which operate on more than one type of object. As ML has no subtyping, it is impossible to create a List which can contain any type. What you can do, however, is define a List implementation which has a type parameter. The type parameter cannot be used to change the behavior of the implementation in any way, and also cannot be restricted in any way. It can, however, be used to define an implementation of a List which can be instantiated into a List of any concrete type.

Generic Java[BOSW98] is an extension to Java which attempts to add ML-style parametric types. This extension only modifies the compiler to allow the compile-time propagation of types through the collections library and other similar APIs. The runtime is not impacted at all, as the extra type information is erased during compilation. This makes the extension both very appealing from a compatibility standpoint, and very limited from a functionality standpoint.

In all the above examples, two parameterized classes with different parameters are unrelated. For example, `List<Number>` is not related to `List<Integer>`, even though `Integer` is a subtype of `Number`. A function that wants to read numbers out of a `List<Number>` cannot accept a `List<Integer>` as an argument. Logically, asking to be able to do this is entirely sensible — if a function is only reading numbers from the list, it does not care what further restrictions there are on the contents of the list. In general, it is not true that `List<Integer>` is a subtype of `List<Number>`, but, as described later, they are related.

The other advantage my system has many of the others is that it is part of a dynamic language. That gives it the ability to do things such as create new parameterizations at runtime, and defer all the type checks until runtime, if necessary. Thus, even if the compiler cannot determine the correct type to give to a unifying type variable at compile time (which is, in general, often hard to do), it can always fall back on the runtime implementation. In a statically typed language, fallback on a runtime type system would generally not be possible, so a failure to analyze the program at compile time would be a problem.

There are some other systems similar to mine. One notable one is described in the paper "Unifying Genericity: Combining the Benefits of Virtual Types and Parameterized Classes"[TT99]. My system similarly has the benefits of structural subtyping and those of virtual types. The major difference is that the described system is implemented for a statically type-checked non-multimethod dispatched language. Implementing a system within the framework of a language like GOO creates some different issues from the ones they deal with. I additionally added the ability to have contravariant subtyping of type parameters, and the ability to use parameters which are not types. I do not however have support for the F-bounded recursive type parameters.

Chapter 3

Specification

3.1 Road map

The specification that follows assumes a basic knowledge of the GOO language. While GOO's syntax is fairly similar to that of Lisp, there are quite a few differences. To help readers who are familiar with Lisp, but may not know about GOO, I have attached a summary of some of the more unique features in Appendix A which you may find useful.

The eventual goal is to be able to declare and use parameterized types, but in order to do so, some more basic functionality must first be present. My system is divided into three sections, each one necessary for the next:

- Type modifiers
- Unification System
- Parameterized Type system

First type modifiers are introduced. These are introduced in order to allow some flexibility in what is required of a parameterized type. Many functions may not care exactly what type a sequence is parameterized on, as long as it's at least restricted to be a number. The type modifiers allow this, among other things.

The unification system and type variables are an essential part of the parameterized type system, and are present in some form in most other implementations (in C++, for instance, as "template parameters"). They allow a function signature to express that two arguments have related types. The `add` method for adding an element to a vector is a good example of a signature that requires this kind of constraint. It must check to ensure that the vector can hold the new element that will be added.

Finally, the syntax to declare, create, and instantiate parameterized types is described. Each of these sections is an important part of the functionality.

3.2 GOO's types

Before getting into the main part of the system I'd like to first discuss the type constructors currently in GOO, and propose a slight change.

GOO has the following type constructors:

- `(t+ type1 type2 ...)` - type union. All types passed as arguments are a subtype of this.
  e.g. `(subtype? <int> (t+ <int> <flo>)) => #t`, and `(isa? 3.1415 (t+ <str> <flo>)) => #t`

- `(t* type1 type2)` - product type. Used for multiple values.
  e.g. `(isa? #(1 2) (t* <int> <int>)) => #t`

- `(t= instance)` - singleton type. Creates a type of which only the one instance is a member.
  e.g. `(isa? 5 (t= 5)) => #t`

- `(t< type)` - subtype-singleton. All types (the types themselves, not instances of those types) which are a subtype of type are a member of this.
  e.g. `(isa? <int> (t< <num>)) => #t`

I first propose to rename the `t=` and `t<` operations to `o=` and `o<`, and they will be referred to by their new names throughout the rest of this paper. These existing operations can be characterized by the relationships in Figure 3-1 in pseudo-GOO. If multiple rules apply, use the first one.

3.3 Type modifiers

The type modifiers allow unique relationships between types that are often inexpressible in a type system. For example, often a function may take a `<vec>` as an argument, but never writes to it. It wants only to read data from the vector, and it requires that the data be a subtype of `<num>` (e.g. `<int>` or `<flo>`). Thus, the passed in vector may be parameterized on any subtype: `<num>`, `<flo>`, or `<int>`. The argument type for this function will be written `(of <vec> type (t< <num>))`.[1]

Another function may take a `<vec>` as an argument, but only *writes* a subclass of `<num>` to it. Therefore, the passed in vector can be parameterized on any *supertype* of `<num>` (namely `<num>`, `<mag>` and `<any>`). This is written `(of <vec> type (t> <num>))`.

Yet another function may wish to read and write, and thus requires that the passed in `<vec>` be parameterized on *exactly* `<num>`. In this case, the type is written simply `(of <vec> type <num>)`, or `(of <vec> type (t= <num>))`, which is identical.
To enable this functionality, I propose the addition of the the following type modifiers:

- `t=` - type equivalence: `(isa? 5 (t= <int>))`, but not `(isa? 5 (t= <num>))`
- `t<` - subtype: `(isa? 5 (t< <num>))` and `(subtype? <int> (t< <num>))`.
- `t>` - supertype: `(subtype? <num> (t> <int>))`

When used with `subtype?`, the `t<` operator is not useful, as it is essentially a no-op. However, a new type comparator called `istype?` is introduced to embody the question that must be asked when comparing type parameters of a parametric type.

For example, when asking if an object of type `(of <vec> type <int>)` is a subtype of `(of <vec> type <num>)`, the correct answer is “no”, as a vector of integers cannot contain floats, but a vector of numbers can. Thus, clearly, the types `<int>` and

[1]The `of` syntax is introduced on page 29.

```
(subtype? (t+ t1 t2 ... t#) x)
  => (and (subtype? t1 x) (subtype? t2 x) ... (subtype? t# x))
(subtype? x (t+ t1 t2 ... t#))
  => (or (subtype? x t1) (subtype? x t2) ... (subtype? x t#))

(subtype? (t* t1 t2) (t* t3 t4))
  => (and (subtype? t1 t3) (subtype? t2 t4))
(subtype? (o= l|<tup>) (t* t1 ... t#))
  => with l as (v1 ... v#):
       (and (subtype? (o= v1) t1) ... (subtype? (o= v#) t#))

(subtype? (t* t1 t2) x)
  => #f
(subtype? x (t* t1 t2))
  => #f

(subtype? (o= x|<class>) (o< t))
  => (subtype? x t)
(subtype? (o< t1) (o< t2))
  => (subtype? t1 t2)

(subtype? (o= x) (o= y))
  => (== x y)
(subtype? (o= x) t)
  => (subtype? (class-of x) t)

(subtype? x|<class> y|<class>)
  => (member? y (class-ancestors x))

(subtype? x y)
  => #f

(isa? x t)
  => (subtype? (o= x) t)
```

Figure 3-1: Subtype rules for standard GOO types

`<num>` cannot be compared with `subtype?`. Without the added type modifiers, the correct question to ask when comparing parametric type parameters is type equality. This is easily defined as:

```
(df eqtype? (x y)
  (and (subtype? x y) (subtype? y x)))
```

With the addition of the `t<`, `t>`, and `t=` modifiers, type equality is no longer the correct question to ask, as those modifiers are supposed to *override* the default of equality, and provide other options for the comparison.

The `istype?` is defined in Figure 3-2 with the new type modifiers. As before, if more than one rule matches, apply the first one.

The `subtype?` method is also extended so that if one of the new type modifiers is passed to it, it calls through to `istype?`. This has the effect of causing `subtype?` and `istype?` to be exactly equivalent, except when no type modifier is present on the second argument. In that case, `istype?` behaves as if `t=` were specified, and `subtype?` behaves as if `t<` were specified. Even though `subtype?` calls `istype?`, and `istype?` calls `subtype?`, no circularity can occur, as `istype?` always unwraps the type modifiers before calling `subtype?`.

A piece of the type hierarchy that the new type modifiers combined with the `istype?` method create is shown in Figure 3-3. Some examples of method calls from that hierarchy are:

- `(istype? <int> <int>) => #t`
- `(istype? (t< <int>) (t< <any>)) => #t`
- `(istype? <int> (t> <any>)) => #f`
- `(istype? <int> <any>) => #f`
- `(istype? <int> (t< <any>)) => #t`

```
(istype? (t= t1) (t= t2))
  => (and (subtype? t1 t2) (subtype? t2 t1))

(istype? (t= t1) (t< t2))
  => (subtype? t1 t2)
(istype? (t< t1) (t= t2))
  => #f
(istype? (t< t1) (t< t2))
  => (subtype? t1 t2)

(istype? (t= t1) (t> t2))
  => (subtype? t2 t1)
(istype? (t> t1) (t= t2))
  => #f)
(istype? (t> t1) (t> t2))
  => (subtype? t2 t1)

(istype? (t> t1) (t< t2))
  => (istype? t2 <any>)
(istype? (t< t1) (t> t2))
  => #f

;; from here out creates the implicit t= when a type without a
;; modifier is passed in because istype? defaults to type equality

(istype? (t= t1) t2)
  => (istype? (t= t1) (t= t2))
(istype? t1 (t= t2))
  => (istype? (t= t1) (t= t2))

(istype? t1 (t< t2))
  => (istype? (t= t1) (t< t2))
(istype? (t< t1) t2)
  => (istype? (t< t1) (t= t2))

(istype? t1 (t> t2))
  => (istype? (t= t1) (t> t2))
(istype? (t> t1) t2)
  => (istype? (t> t1) (t= t2))

(istype? t1 t2)
  => (istype? (t= t1) (t= t2))
```

Figure 3-2: Definition of `istype?` and type modifiers

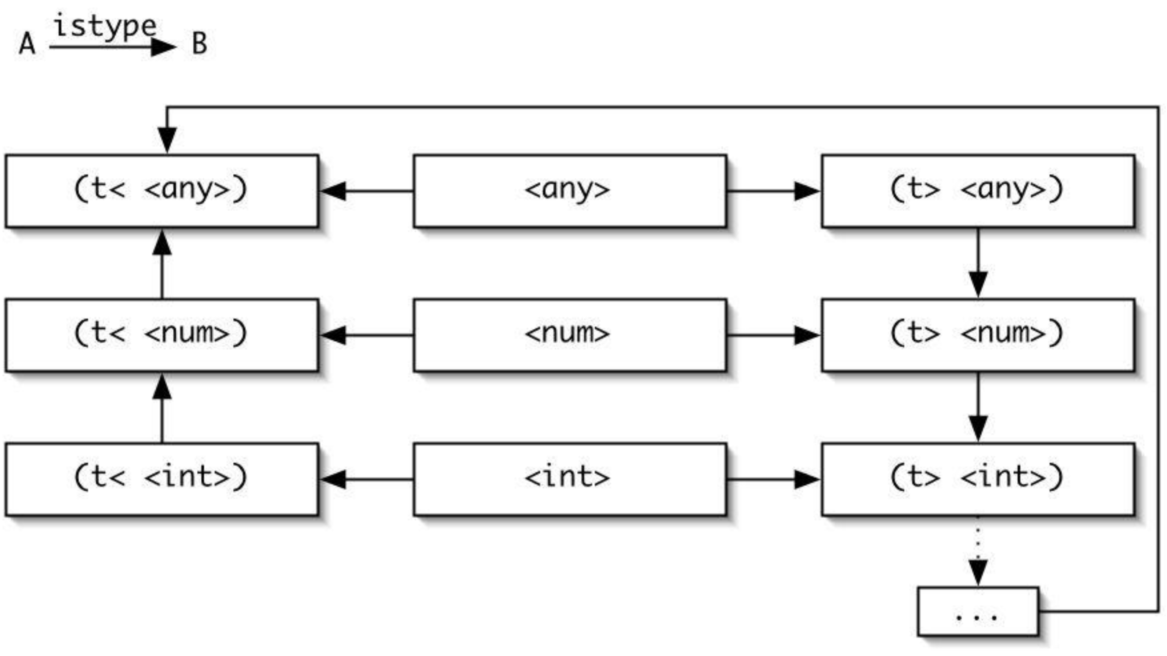

Figure 3-3: A simple type hierarchy illustrating `istype?` relationships

3.4 Unification system

The unification system is an addition to the type checking system that adds the ability to do a type check against an unknown or partially unknown type, and if the type check succeeds, the conditions under which it was possible to succeed are recorded.

The ultimate goal of this addition is to make it possible to express a method signature such as "takes two arguments, 'a' and 'b' which can be any type, and returns a value of the same type as 'a', whatever that may be." or "takes two arguments 'a', and 'b', and 'b' must be an instance of 'a'."

I propose the new syntax:

```
(? name)
```

This creates a type variable whose scope is the current function. One can do type checks against it, e.g.

```
(isa? 3 (? t))
(subtype? (? t) <str>)
```

The type variable starts out with no restrictions. After each type check, the range of possible values the type variable can take on is recorded. Taking the above example, `(isa? 3 (? t))` would succeed and return true, as this is the first mention of `(? t)`, and thus it has no restrictions. Then, `(? t)` is restricted such that from then on, it can only take on a type that would not violate the check `(isa? 3 (? t))`. This implies the constraint "`t` is a supertype of `<int>`", or, in GOO, one would say `t` must be a supertype of `<int>`. Thus, `<int>`, `<num>`, and `<any>` would be the simple types that `t` is restricted to. Of course, complex types such as `(t+ <int> <str>)` are also possible.

Now, executing the second check, `(subtype? (? t) <str>)` will check the previous restrictions on `t` (that it must be a supertype of `<int>`), and then return false, as there is no possible way for `(? t)` to be both a supertype of `<int>`, and a subtype of `<str>`. In this case, as the type check failed, no restrictions are added to `(? t)`.

Executing `(isa? "some string" (? t))` would return true, as `(? t)` *can* be a supertype of both `<str>` and of `<int>`, namely the type `<any>`, or the type `(t+ <int> <str>)`. As a side-effect of the type check succeeding, `t` will get further restricted to this type.

When checking `subtype?` with a type variable, ensure that there is a type for the type variable such that all past checks will remain true and the current check returns true as well. If the new constraint can be added, return true, otherwise, return false and do not modify the type-variable state. These rules can be concisely expressed in pseudo-code as shown in Figure 3-4.

Type variables can also be used in a function signature. The easiest way to think about the semantics of this is to pretend that the signature checking occurs inside the function body and translate the type check of the signature to code in the body.[2] The scope of the type variable is the entire body of the function, plus the signature check, plus the return type check. Note that when I say the scope includes the function body, I mean just that: you may use the type variable inside the function. I do *not*

[2]This translation only works for non-generic functions, so it cannot be used in practice, but is useful for descriptive purposes

```
(subtype? int num)
(typevar-possible? tv a)
  => (there exists a type "t" for tv s.t.
       (for all restrictions on tv
         (if (== (restriction-type restriction) 'subtype)
           (subtype? t (restriction-val restriction))
           (subtype? (restriction-val restriction) t)))

(subtype? (? tv) a)
  => (seq
       (add-restriction tv 'subtype a)
       (typevar-possible? tv))

(subtype? a (? tv))
  => (seq
       (add-restriction tv 'supertype a)
       (typevar-possible? tv))

(subtype? (? tv1) (? tv2))
  => (error "Cannot compare type variables")

Place this wrapper around all subtype? calls:
(subtype? t1 t2)
  => (seq
       (def old-tv-state (dup *tv-state*))
       (def result (subtype? t1 t2))
       (if (== result #f)
           (set *tv-state* old-tv-state))
       result)
```

Figure 3-4: Subtyping rules for type variables

mean that a use of the type variable in the function will affect whether the function is applicable to a set of arguments. Doing so would require analyzing the body of the function which is undesirable as well having questionable usefulness.

An example of this:

```
(df return-self (v|(o= (? x)) => (o= (? x)))
    v)
```

becomes

```
(df return-self (v)
    (if (not (isa? v (o= (? x))))
         (error "Invalid type")) ;; never happens in this case
    ;; done with argument checks, now run the code in the method
    (def result v)
    ;; done with body, now check result type
    (if (not (isa? result (o= (? x))))
        (error "Invalid return type")))
```

This has the expected meaning: the method can take any argument, and it returns the argument itself.

3.4.1 Interaction with type constructors

So far, I've only described type variables used in isolation. However, they can also be combined with any of the other type modifiers and constructors.

The interaction of type variables and `t+` is especially interesting, as the order of the arguments to `t+` is now significant. Take the following example:

`(isa? 5 (t+ (? x) <int>))`

This translates to essentially: `(or (isa?  5 (? x)) (isa? 5 <int>))`

Now compare that with:

`(isa? 5 (t+ <int> (? x)))`

which translates to:

`(or (isa? 5 <int>) (isa? 5 (? x)))`

At first glance they may seem identical, but remember: `or` will stop evaluating conditions as soon as it finds condition that isn't false, and doing a type check against a type variable creates side effects. Thus, in the first example, `(? x)` will have had the restriction that it is a supertype of `<int>` added, while, in the second case, a type check against `(? x)` was never even performed (the first test in `or` succeeded, so it doesn't evaluate the rest), and thus no restriction was added.[3]

The `t*` type constructor is akin to `and`, and thus does not cause the same issue. If the type check succeeds, then all of the sub parts must have succeeded. If one of the sub parts did not succeed, the type check fails, and thus, none of the type variables are modified (partial side-effecting is not allowed - the implementation must make sure either all the modifications are done, or none are.).

The `o=` operator has a dual role: it can either be used with a type as its argument, or with a non-type. For example `(isa? <int> (o= <int>))`, and `(isa? 5 (o= 5))`. In the case where it is used with a type, `(isa?` x `(o=` y`))` is equivalent to `(istype?` x y`)`. Thus, if x is a type, and y is a type variable, it will just work out.

However, if x is not a type, then the type variable must take on a value which is not a type as well. Therefore, when evaluating a condition like `(isa? 5 (o= (? t)))`, `(? t)` will be restricted to be exactly `5`. If a type variable is set to a value which is not a type, and then used in a position which requires it to be a type, the type check will always fail.

The `o<` operator can only take a type as its argument, so there are no complications with type variables. The expression `(isa?` x `(o<` y`))` translates to `(subtype?` x y`)` via the previously defined rules in Figure 3-1.

The type modifiers `t<`, `t>`, and `t=` do not affect the type variable code, as each type modifier is implemented in terms of an appropriate more primitive `subtype?` call. Thus, the type variable code can be blissfully ignorant that the type modifiers even exist.

[3] I'm not entirely sure if these semantics are the most sensible, but it follows from the standard definition of `or`. One possible solution to this problem is to restrict the `t+` operator to only allow one type variable, and always attempt to type check against the base types first.

3.4.2 Dispatcher

As briefly stated above, the ultimate purpose to this extension is to make the generic method dispatcher more powerful. Type variables are not particularly necessary in a method body, as you can easily write equivalent code without them. The main advantage is as an extension to the method dispatcher system. However, inclusion in the dispatcher is not as simple an extension of the stand-alone semantics as might be hoped. To understand how this extension to the type system behaves, it will help to first know how GOO's standard dispatcher works.

GOO currently has a fairly standard multimethod dispatch system. This provides the ability to declare multiple functions with the same name and differing parameter types. Unlike languages such as C++ and Java, where only the first argument (the implicit 'self' argument) is dispatched on at runtime, in a multimethod dispatch system, all the arguments are used at runtime in determining which of the multiple methods to call. No behavior is determined from the declared type of a variable being passed to the function, only the actual runtime type. Dispatch consists of two steps:

1. finding all the methods applicable to the actual arguments, and

2. finding the most specific of those methods.

For example, take a generic with three methods signatures (the function body is not relevant to dispatch):

```
(dm + (a|<num> b|<num> => <num>) ...) ;; #1
(dm + (a|<flo> b|<flo> => <flo>) ...) ;; #2
(dm + (a|<int> b|<int> => <int>) ...) ;; #3
```

When executing a call `(+ 1 2)`, it would first figure out which signatures are consistent with the arguments, using the `isa?` method. 1 is a `<int>`, and `<int>` is a subtype of `<num>`, so both methods #1 and #3 are applicable.

This algorithm to test the applicability of a method is essentially:
For every actual argument `x` and specified type `t`, ensure `(isa? x t)`.

The second step is to test which method is most specific. The general idea is that method A is more specific than method B, if A is applicable to a pure subset of the values B is applicable to.[4] In this case, we have to decide between method 1 and method 3. As `<int>` is a subtype of `<num>`, method 3 will be more specific, and thus the method that will be called.

The algorithm for this is approximately:

```
(order-specs (T1 T2)):
  (if (subtype? T1 T2)
      (if (subtype? T2 T1)
          '=
          '<)
      (if (subtype? T2 T1)
          '<
          '<>))

(compare-methods (M1 M2)):
  (def spec-ordering
    (map order-specs (argument-types M1) (argument-types M2)))

  If all elts of spec-ordering are '= or '<:
    M1 more specific
  Else If all elts of spec-ordering are '= or '>:
    M2 more specific
  Else:
    Ambiguous
```

Then, find the one method which is more specific than all the others, by sorting the methods based on the compare-methods function. If there is no one methods which is most specific because of ambiguous methods, an error occurs.

This may seem much more complex and costly at runtime than single argument dispatch, and it can indeed be. However, in a sophisticated system, much of the overhead will be removed at compile-time as the optimizer can often pre-sort the methods, and eliminate choices that are never possible given some amount of information about the argument values at a certain call site. Thus, the cost can be minimized, and the additional functionality is worth the slight slowdown.

[4]That is not entirely true, as multiple inheritance introduces a slight twist, but, for the most part it is a reasonable approximation

3.4.3 Interaction with dispatcher

Type variables

In the specification for how to evaluate type-variables in a function signature above, I stated that you can think of the type checks as being part of the function body. While this actually works as an implementation strategy for normal functions, this will not work for a generic method, as the dispatcher has to check method applicability *before* calling the function. To make matters even worse, it has to determine method applicability for possibly *many* functions, before figuring out which is most applicable. As the semantics require that the scope of the type variable includes the method body and the return type check, the type-variable state information must be stored for every method the dispatcher might possibly call. Then, when the call occurs, that information must be passed in to the method.

Method sorting

Checking whether one method is more specific than another when type variables are involved is somewhat more complex. For example, which is most specific:

```
(dm test (a|<int>) ...) ;; #1
(dm test (a|(t= (? y)) ...) ;; #2
```

Trying to use the same algorithm that the normal dispatcher uses would yield the following: the first method is more specific if

```
(subtype? <int> (t= (? y)))
```

Clearly there is a binding for `(? y)` that will succeed, namely `<int>`. However, the same claim can be made when checking if the second method is more specific. Checking `(subtype? (t= (? y)) <int>)` will also return true and bind `(? y)` to `<int>`. Thus, neither appears more specific than the other.

However, this is not the correct answer. To correctly sort the methods, the following procedure should be followed:

- First, given the actual parameters to the method call unification should be performed on each method separately.
- Now the possible values for each type variable are known. Keeping the type variables constant, apply the usual method sorting rules.

For example, if the above method `test` is called with an argument of `1`, after type variable substitution, the method signatures become: `(a|<int>)` and `(a|(t= <int>))`. Since `(t= <int>)` is more specific than `<int>`, the second method wins. In this particular case, method #1 is actually never going to be the most applicable.

Note that some methods cannot be sorted. For example:

```
(dm foo (a|(t= (? x)) b|(t= (? y))))
(dm foo (a|(t= (? z)) b|(t= (? z))))
```

You might expect that the second method is more specific than the first, but, if you actually call `(foo 1 1)`, both methods will be equally applicable, as the type variables in both cases will be `<int>`. Calling the method with arguments of different types, like `(foo 1 1.2)` will make the first method the only applicable method, and thus the most applicable.

3.5 Parameterized type system

3.5.1 Creating parameterized types

There is no special syntax to declare a parameterized class. One simply declares a regular class and adds parameters to it. Using the base class type on its own is equivalent to instantiating the type with no parameters specified.

To add a parameter slot to a class:

`(dparam ,name (,owner => ,type) ,default)`

This creates a parameter for class owner called name. All existing instances of owner get the parameter set to default. Like `dp` (the syntax for defining a property on an object) it adds a method to the generic name. However, *unlike* `dp`, the method is applicable to the owner class itself and concrete parameterizations of the class, *not* instances of the class. This slot is read-only, as mutating the type parameter would change the type of the object, and that is not allowable.

For example, to add parameters to the GOO hashtable type:

```
(dparam type (<tab> => <type>) <any>)
(dparam hash-fun (<tab> => <fun>) id-hash)
(dparam eq-fun (<tab> => <fun>) ==)
```

This creates a `type` parameter of type `<type>` to specify the type of the values in the hashtable, a `hash-fun` parameter of type `<fun>` to specify what hashing-function to use, and a `eq-fun` parameter of type `<fun>` to specify what equality function to use. The defaults are, respectively, `<any>`, `id-hash`, and `==`. Note that parameters are not required to be types, but can be any object.

3.5.2 Instantiating a parameterized type

To instantiate a parameterized type with specified parameters:

```
(of base-class|<parambase> parameter-inits|...)
```

where parameter-inits is groups of `parameter|<gen> parameter-value|<any>`. For example, to create a hashtable which will contain integers, and has a custom hash function:

```
(of <tab> type <int> hash-fun my-hasher)
```

This is similar to the syntax for `new`, but creates a new class, not a new instance. The syntax is designed to look similar to keyword syntax, if GOO had such a thing.

In this example, the `eq-fun` parameter is not specified. Creating a type with some parameters unspecified is allowable and extremely useful. Parameters which are not specified in the call to `of` are treated as a wildcard for the purposes of type checking. Thus, if you parameterize `<tab>` with only `type` specified, a table with any value for `eq-fun` and `hash-fun` will be a subtype of it.

Note that calling `of` multiple times with equal (`==`) arguments will return the same type twice, and will not construct a unique type each time.

To create an instance of the new parameterized type you could do:

```
(dv my-table (new (of <tab> type <int> hash-fun my-hasher)))
```

Any unspecified parameters are given the default value when the object is instantiated. For example,

```
(new (of <tab> type <int> hash-fun my-hasher))
```

is equivalent to

```
(new (of <tab> type <int> hash-fun my-hasher eq-fun ==))
```

The lack of a strong difference between parameterized types and non-parameterized types is to allow simple interoperability with non-parameterized GOO code. If user code uses the `<tab>` class and completely ignores the fact that it is parameterized, it will work in most cases just as if parameterized types did not exist. Creating a new `<tab>` will use the defaults, and doing a type check against `<tab>` will succeed no matter what the parameter arguments are. The only place ignorance of parameterized types breaks down is if you need to call a library function that takes a restricted type, or if you want to make your own type in the collections hierarchy.

3.5.3 Mutable types

To ensure program correctness, the type system usually guarantees that any type check on an object will be forever valid. It is not possible to mutate an object's class so that something you previously thought was a mutable list is now an immutable list, for instance. If this was possible, then variables with restricted types could be confused into holding an incorrect type by mutating it after the object reference was inserted into the variable.

The parameterized type system does prevent you from modifying any of the type parameters directly by making them read-only. However, it is possible to pass arbitrary objects to `of`, such as a mutable array. Unfortunately, there is no way in the GOO type system to specify that a given parameter must be immutable. Thus, it is possible to confuse the type system badly if you do indeed use a mutable array as a type parameter and then mutate it. All I can say is: you must not mutate objects used as parameters, as doing so could confuse the type system and cause all sorts of bad things to happen.

3.5.4 Type relationships

Parameterized types have the following type relationships:

- `(of base-class ...)` is a subtype of base-class.
- `(of base-class some-param1 value1 some-param2 value2 ...)` is a subtype of
  `(of base-class some-param1 value1 ...)`
- `(of base-class some-param value1)` is a subtype of
  `(of base-class some-param value2)` ⇔ `(istype? value1 value2)`
- `(of base-class1 some-param value)` is a subtype of
  `(of base-class2 some-param value)` ⇔ `(subtype? base-class1 base-class2)`.

Taking the "my-table" example from above:

```
(dv my-table (new (of <tab> type <int> hash-fun my-hasher)))
(isa? my-table <tab>) => #t
(isa? my-table (of <tab> type <int>)) => #t
(isa? my-table (of <tab> type <num>)) => #f
(isa? my-table (of <tab> type (t< <num>))) => #t
```

Chapter 4

Examples

The following example introduces a simplified version of the GOO collections hierarchy. First the basic types are introduced, and then some essential methods are defined for the types. Note that `<chr-vec>` can be defined in order to create a more compact encoding for that particular case.

```
;; a collection is some kind of key->value mapping
(dc <col> (<any>))
(dparam key-type (<col> => <type>) <any>)
(dparam val-type (<col> => <type>) <any>)

;; a sequence is a collection with contiguous nonnegative integer keys
(dc <seq> ((of <col> key-type <int>)))

;; Vector - a concrete type of sequence
(dc <vec> (<seq>))

;; an abstract type which can hold arbitrary key->value mappings
(dc <map> (<col>))

;; Hashtable - a concrete type of map
(dc <tab> (<map>))
(dparam eq-fun   (<tab> => <fun>) ==)
(dparam hash-fun (<tab> => <fun>) id-hash)

;;; now add some methods..

(dm elt (    c|(of <vec> val-type (t< (? x)))
         index|<int> => (? x))
```

```
  ...)

(dm elt (  c|(of <tab> val-type (t< (? x)) key-type (t> (? y)))
         key|(? y)
         => (? x))
  ...)

(dm elt-setter (  val|(? x)
                    c|(of <vec> val-type (t> (? x)))
                index|<int>)
  ...)

(dm elt-setter (val|(? x)
                  c|(of <tab> val-type (t> (? x)) key-type (t> (? y)))
                key|(? y))
  ...)

(dm add! (c|(of <vec> val-type (t> (? x))) e|(? x))
  ...)

(dm add-all! (     col|(of <col!> val-type (t> (? t)))
              new-elts|(of <col>  val-type (t< (? t))))
  (for ((e new-elts))
    (add! col e)))


(dg fill (col|(of <col> val-type (t> (? t)))
            x|(? t)
          => (of <col> val-type (? t)))
  ...)


(dc <chr-vec> ((of <vec> val-type <chr>)))

;; special elt setter and getter for <chr-vec> which optimizes storage.
(dm elt (c|(of <vec> val-type <chr>) index|<int> => <chr>)
  ...)

(dm elt-setter (val|<chr> c|(of <vec> val-type <chr>) key|<int>)
  ...)

(dm new (parent|(o= (of <vec> val-type <chr>)) inits|...)
  (app new <chr-vec> inits))
```

Now that a set of basic operations is defined, some examples of the usage:

```
(dv x (new (of <vec> val-type <num>)))

(subtype? x (of <vec> val-type <any>))
  => #f

(subtype? x <vec>)
  => #t

(subtype? x (of <vec> val-type (t< <any>)))
  => #t

(subtype? x (of <col> key-type (t< <num>)))
  => #t

(add x 5)
(add x 6.0)

(elt x 0)
  => 5

(elt x 1)
  => 6.0

(add x "foo")
  => ERROR: no applicable method

(df foo (v|(of <vec> val-type (t< <num>)) => <num>)
  (+ (elt v 0) (elt v 1)))

(foo x)
  => 11.0
```

Chapter 5

Implementation

5.1 Overview

The goal of this project was not to create an efficient implementation, but rather, to create a set of reasonable and powerful semantics. However, to be useful, there must be some hope of a fast implementation, and I will discuss that afterwards.

GOO is an evolving project, and thus it is likely that some form of this system will eventually make it into the core of the language, and be used in core libraries like the collections. However, in order for the system to be used to implement the core functionality like `<vec>`, `<lst>`, etc., this code has to not depend on that same functionality it is implementing. To avoid all these issues, and make the initial design of the code simpler, I have written the code in high-level GOO, and made as few modifications to the core of GOO as possible. This makes it a lot more straightforward and easier to understand, but it does introduce a number of compromises. I will note these discrepancies between the implementation and the specification as they come up. It is expected that at a future point in time, this implementation can be rewritten in low-level GOO, and integrated into the language.

The implementation for the new type modifiers consists of defining new functions for the type modifiers, and defining the correct `istype?`, `subtype?`, and `eqtype?` methods to operate on them. The code is quite simple, and follows directly from the definition of `istype?` shown in Figure 3-2.

The type variable changes are more substantial, and includes the `(? t)` macro, functions to calculate the intersection and union of types, new methods on `subtype?` to side-effect the type variables, and hooks into dispatch to coordinate the type variable state.

The last section, parametric types, introduces the `dparam` and `of` syntax, and more extensions to `subtype?` to support the relationships between parameterized types. After all this, you can finally build a describe a method which takes an object and a vector which must be able to contain that type.

5.2 Type modifiers

First the three new type modifiers need to be declared. The type modifiers are themselves subclasses of the type `<typemod>`:

```
(dc <typemod> (<type>))
  (dp type-val (<typemod> => <type>))
(dc <type-equal> (<typemod>))
(dc <type-sub> (<typemod>))
(dc <type-super> (<typemod>))
```

The functions `t=`, `t<`, and `t>` are simply constructors for those types.

The major change to the type system is implementing the `istype?` method to operate on these new types. `istype?` is in turn implemented by a combination of `subtype?` and `eqtype?`. The `subtype?` method is canonical except in the case where either argument is a `<typemod>`, in which case it calls through to `istype?`.

Here is a small sample of the methods defined for `istype?`; the rest are similar:

```
(dm istype? (t1|<type-super> t2|<type-sub> => <log>)
  (eqtype? (type-val t2) <any>))
(dm istype? (t1|<type-equal> t2|<type-sub> => <log>)
  (subtype? (type-val t1) (type-val t2)))
(dm istype? (t1|<type-sub> t2|<type-equal> => <log>)
  #f)
(dm istype? (t1|<type-equal> t2|<type-super> => <log>)
  (subtype? (type-val t2) (type-val t1)))
(dm istype? (t1|<type-super> t2|<type-equal> => <log>)
  #f)
```

The `eqtype?` function is implemented simply by checking if both types are subtypes of each other. In a future version, a more efficient version of this method can be written for specific cases.

```
(dm eqtype? (t1|<type> t2|<type> => <log>)
  (and (subtype? t1 t2) (subtype? t2 t1)))
```

The `subtype?` function is extended with the following methods:

```
(dm subtype? (t1|<type> t2|<typemod> => <log>)
  (istype? t1 t2))
(dm subtype? (t1|<typemod> t2|<type> => <log>)
  (istype? t1 t2))
(dm subtype? (t1|<typemod> t2|<typemod> => <log>)
  (istype? t1 t2))
```

to pass all forms with a type modifier as an argument to `istype?`.

This implementation does not currently handle singleton (`o=`) and subclass (`o<`) types properly. Currently in GOO it is assumed that singletons cannot appear nested inside of arbitrary types, and this assumption is hard coded into the method dispatcher. Therefore, without doing some significant work to fix the method dispatcher, I must currently omit support for singleton types.

5.3 Unification system

5.3.1 Type variables

Every type variable consists of two types: the maximal type it can take on, and the minimal type. An unrestricted type variable has as its minimal type `<any>` and its maximal type a new type `<type-bot>`. The `<type-bot>` type is the analog of `<any>` at the other side of the hierarchy: it is a subtype of every other type. This type is used only to denote no restriction on the maximal type of a type variable. No object is an instance of `<type-bot>`. The functions `typevar-min-type` and `typevar-max-type` return the minimal and maximal types for a type variable.

As type checks are performed on these type variables, additional restrictions are added to them. These restrictions are done by determining the intersection or union of the tested against type with either the maximal type, the minimal type, or both.

5.3.2 Adding restrictions

The union of type types is determined by the `type-union` function. The union of two types can be very simply represented just by the result of applying `t+` to them. This function tries to be a little smarter than that in order to minimize the size of the union type as much as possible. Thus, it first checks to see if either type is completely enclosed by the other and returns it if so.

```
(dm type-union (t1|<type> t2|<type> => <type>)
  (if (subtype? t1 t2)
      t1
      (if (subtype? t2 t1)
          t2
          (t+ t1 t2))))
```

More methods could be added to the type-union generic to eliminate common terms if one of the arguments is itself a union type which is partially subsumed by the other type.

The intersection is determined by the `type-intersection` method. Intersecting two types is slightly more complicated. When dealing simply with classes, this is an easy problem, as the inheritance tree is explicit. All it has to do is get a list of all the ancestors of both classes, and find all the common elements. Then, take the union of those.

However, when dealing with non-class types, the algebraic relationship to other types is not explicitly known to the system. Given any two types, `subtype?` will tell you their relationship, but there is no way to *search* for all the common types and find the most specific one. Also, unlike the union operation, the intersection cannot simply return an “intersection type” as no such thing exists. As the intersection should always be representable without one, no such type need exist. However, this means that methods for `type-intersection` must be created for all the types in the system in order to encode the knowledge of their relationships.

One mitigating factor is that `type-intersection` will never get called for any of the type-modifier types, as those have been transformed ahead of time into standard `subtype?` calls. It will also never get called with a type-variable type, as it is not allowable for one type variable to depend on another. Thus, methods must be written for `<class>`, `<singleton>`, `<subclass>`, and `<union>`.

5.3.3 Subtype rules

With `type-intersection` and `type-union` declared, the implementation of `subtype?` is relatively simple. There are three interesting base cases:

1. checking if a normal type is a subtype of a `<typevar>`:
   `(dm subtype? (t1|<type> t2|<typevar> => <log>) ...)`
2. checking if a `<typevar>` is a subtype of a normal type
   `(dm subtype? (t1|<typevar> t2|<type> => <log>) ...)`
3. checking if a `<typevar>` is a subtype of another
   `<typevar>`:
   `(dm subtype? (t1|<typevar> t2|<typevar> => <log>) ...`

In the first case, `t1` must be a subtype of the minimum value for the typevar `t2`. If it is, the maximum for `t2` is updated to `(type-intersection (typevar-max-type t2) t1)`. In the second case, the maximum value for the typevar `t1` must be a subtype of `t2`. If it is, the minimum for `t1` is updated to `(type-union (typevar-min-type t1) t2)`. The third case causes an error, as it is not allowable per the specification to compare two type variables.

5.3.4 Issues

To implement type variables as fully specified requires that the method dispatcher be able to pass the type variable state into the body of the method. However, there is currently no mechanism by which to accomplish this in GOO.

I expect this to eventually be done through some sort of auxiliary parameter functionality, which the method dispatcher can pass the values of the type variables through. Then, the type variables would be stored in the function's activation frame like a normal local variable, until the return type is checked and the frame is destroyed.

However, as I did not wish to get too deep into the core of GOO, I opted for a much simpler approach at the moment: the value of the type variables is simply kept in a global variable "`*typevar-states*`". As there is no simple way to tell the compiler to make all mentions of `(? t)` in one function and its specializers point to the same object, the type variable objects themselves do not point to storage - they are simply names that are looked up in the "`*typevar-states*`" table. This means all mentions of `(? t)` will have the same storage globally, instead of each function's `(? t)` being separate. Thus, to make dispatch work, the values are all reset for each method applicability check.

Therefore, currently, using type variables inside a method body will give you arbitrary and incorrect results, but they function in the signature. For the same reason, the return type check is inoperative.

5.4 Parameterized type system

The parameterized type system extends the type "`<class>`" to have new attributes:

- `class-direct-params` contains the parameters declared on this class itself.
- `class-params` contains the parameters declared on this class or one of its ancestors.
- `class-param-value` contains the values of the parameters defined for this class.
- `class-param-default` contains the default values for parameters used when an explicit value was not provided.

A small amount of complication arises in the present implementation, as it is not possible to cleanly subclass `<class>`, and thus, I could not add attributes directly on it. This is a minor issue, however, and does cause any serious problems.

The parameter declaration syntax `(dparam name (owner => type) default)` is a straightforward macro which adds the parameter to `class-direct-params` and creates a method "name" which retrieves the value of the parameter on a class.

The syntax to create a parameterization of a type, `of`, essentially creates a new subclass of the parameter base class with the specified values filled in the parameter slots. However, this new class is actually of type "`<param>`", a subclass of the "`<class>`" type.

The subtype rules for parameterized types are simple:

- First check if the first type is a declared *subclass* of the second. This would be the case for instance in `(subtype? (of <vec>) <vec>)` because `of` creates the new type as a subclass.

- If that fails, make sure the "parameter base" of the first is a subclass of the parameter base of the second. Parameter base is defined as the first parent class which is not defined with `of` and thus does not follow parameterization subtyping rules.

- Next, if that succeeded, make sure that for every parameter declared on the second type, the first one has a compatible parameter value. Compatible means that if the parameters are `<type>`s, comparing them using `istype?` will return true, or, if the parameters are not `<type>`s, that they are equal.

The algorithm can be somewhat more clearly explained with the code:

```
(dm param-subtype? (t1 t2)
  (or (subclass? t1 t2)
      (and (subtype? (param-base t1) (param-base t2))
           (all? (fun (f)
                   (param-compat? (f t1) (f t2))
                 (class-params t2)))))
```

The major issue with the current implementation of parameterized types is that they cannot be used in the signature for most of the standard GOO methods. Thus, defining a new parameterized vector type cannot actually be done unless all the methods are given a different name from the standard GOO name (for instance, prefix every method name with `p-`). This problem occurs because all of the code for dealing with parameterized types, type variables, and modifiers depends on the core generic methods such as `elt`, `add`, `del`, etc. If a method containing a parameterized type in its signature is added to `elt`, then an infinite loop will result, as the code to determine whether that method is applicable will itself call `elt`.

This problem can be solved by writing the code to not depend on any generic method dispatch except in tightly controlled conditions that are guaranteed to not cause problems. It also could possibly be solved by significantly modifying GOO and making it use a protected layer model, where nothing done in a higher-level layer can break code running in a lower-level layer. Then, the dispatch done inside the parameterized type code will use the lower-level dispatcher and thus bypass the possibility for an infinite loop.

Chapter 6

Future Directions

6.1 Additional Constraints

Currently my system does not allow type variables to be bound by by static constraints, but only by passed-in variables. It can be useful to supply additional constraints on the type variable. For instance, take the following method:

```
(dm rest (col|(? x) => (? x))
  (sub col 1 (len col)))
```

Perhaps you want to make a specialized version of this for subtypes of `<num>`. You want to be able to do something like:

```
(dm rest (col|(? x) => (? x) where (? x)|(t< <col>))
  (sub col 1 (len col)))
```

This system does not yet provide such a mechanism, although it ought to in a future implementation. There is no core problem preventing such constraints from being added, just a lack of appropriate syntax in GOO to be able to express it.

6.2 Function Types

One feature that is related to parameterized types is function types. It is often useful to be able to specify the required signature of a function. For example, take the `map` function, which takes a function and a collection, and returns the result of applying the function to each element in the collection. The current signature is:

```
(dg map (f|<fun> x|<col> => <col>))
```

A signature such as the following would be appropriate:

```
(dg map (f|(of <fun> specs (t* (t> (? x))) => (t< (? y)))
         x|(of <col> val-type (? x))
 => (of <col> val-type (? y)))
  ...)
```

This would precisely define the function which map takes and its relationship with the collection passed in and the collection returned. Notice that it requires very few changes to the parameterized type system, as the multiple types in the signature can be easily represented by the `t*` operator, and the relationships are defined using the existing type variables.

6.3 Method ordering

One thing that is somewhat bothersome is that, under my system, method ordering on a generic method can change depending on the actual arguments passed in. This occurs because the `t>` modifier changes the sort order. Having a non-static ordering is somewhat undesirable, as it will reduce the compile-time optimization capabilities. It may perhaps be possible to come up with some set of restrictions on when the modifiers may be used in order to bypass this problem.

The type union `t+` rule is also similarly bothersome, as `(t+ <num> (? x))` and `(t+ (? x) <num>)` are very different types. One could restrict `t+` to only allow a type variable on one branch, and always evaluate that branch last. With that restriction, ordering under `t+` will again cease to matter. As I was not sure how much of a problem this would pose I did not propose that solution in the main body of the paper.

6.4 Parameterized Generics

So far I have not touched upon the concept of the generic function signature itself (as opposed to the signature of the methods *on* the generic function. The signature of the generic function restricts the signatures of the methods added to it such that they must be more constrained. How this applies to signatures with type variables is not completely clear.

6.5 Static Analysis

The two major reasons for wanting to have more expressive types in a language are: 1) safety, and 2) increased speed. However, it is a perverse fact that in a dynamically type language, adding additional types can actually slow down the program. With an unoptimizing compiler, every time a type is used, the compiler must check the actual value against the specified type. All these type checks can cause significant slowdowns. Unfortunately, the compiler cannot just ignore the types at it's leisure, as it must guarantee that the type constraints are not violated.

Adding types can dramatically improve the speed, however, when the compiler uses the additional information to eliminate some type checks and precompute the appropriate method of a generic function to call. Adding parameterized types and type variables magnifies both the good and bad effects. Because there is more type information available, the compiler has more concrete information on which to base its analysis. However, if the analysis fails, the speed hit is even higher.

One situation where parameterized types really help out is analyzing collections operations. It is often infeasible to analyze the entire program and determine that a certain `<vec>` will only ever contain `<int>`s. However, if the programmer specifies that, then the compiler knows immediately what method to call for `+` when running something like `(+ (elt v 0) (elt v 2))`.

6.6 Dynamic Behavior

Sometimes, the code cannot be statically analyzed, and type checks and method dispatch must remain at runtime. Thus, it is also very important for the runtime performance to be as fast as possible. One optimization trick that can often be performed is to assume the class hierarchy of the program is going to be mostly static, and pre-generate a custom dispatcher for each function with the minimal number of type checks to find the most applicable method to call.
For example, given:

```
;; elt-met-1
(dm elt (    c|(of <vec> val-type (t< (? x)))
         index|<int> => (? x))
  ...)

;; elt-met-2
(dm elt (  c|(of <tab> val-type (t< (? x)) key-type (t> (? y)))
         key|(? y)
         => (? x))
  ...)
```

a sufficiently smart compiler could create a custom dispatch method as shown in Figure 6-1 which would be very much faster than running the general algorithm.

```
(df elt-dispatch (arg1 arg2)
  (if (isa? arg1 <vec>)
      (seq
        ;; doesn't need to do any type variable checks as that is only
        ;; used for the return value, and if the function needs to check
        ;; it's return value, it can figure out the value of (? x) itself.
        (elt-met-1 arg1 arg2))
      (if (isa? arg1 <tab>)
          (seq
            ;; (? y) can be between the key-type of the <tab> and the type
            ;; of the key.
            (def q-y-min (key-type (class-of arg1)))
            (def q-y-max (class-of arg2))
            ;; if the type-variable range isn't negative
            (if (subtype? q-y-max q-y-min)
                (elt-met-2 arg1 arg2)
                (error "No applicable methods")))
          (error "No applicable methods"))))
```

Figure 6-1: An example of a custom dispatch function

Appendix A

GOO Mini-tutorial

The following is a brief introduction to GOO. It is intended to explain enough of the syntax and operation of the language to give readers already familiar with Scheme and Lisp the background necessary to understand the examples given in the rest of the paper. Most of the material in this section is *directly quoted* from the GOO manual by Jonathan Bachrach, which is available online at `http://www.googoogaga.org/`. For more detailed information, please refer to the manual.

GOO is a dynamic type-based object-oriented language. It is designed to be simple, productive, powerful, extensible, dynamic, efficient and real-time. It heavily leverages features from many earlier languages. In particular, it attempts to be a simpler, more dynamic, lisp-syntaxed Dylan [Sha96] and an object-oriented Scheme. GOO's main goal is to offer the best of both scripting and delivery languages while at the same time incorporating an extreme back-to-basics philosophy.

A.1 Functions

All operations in GOO are functions.

Functions accept zero or more arguments, and return one value. The parameter list of the function describes the number and types of the arguments that the function accepts, and the type of the value it returns.

There are two kinds of functions, methods and generic functions. Both are invoked in the same way. The caller does not need to know whether the function it is calling is a method or a generic function.

A method is the basic unit of executable code. A method accepts a number of arguments, creates local bindings for them, executes an implicit body in the scope of these bindings, and then returns a value.

A generic function contains a number of methods. When a generic function is called, it compares the arguments it received with the parameter lists of the methods it contains. It selects the most appropriate method and invokes it on the arguments. This technique of method dispatch is the basic mechanism of polymorphism in GOO.

All GOO functions are objects, instances of `<fun>`. Generic functions are instances of `<gen>` and methods are instances of `<met>`.

The form `(FUN ,sig ,@body)` creates an anonymous method, where `,sig` ≡ `(,@vars)` or `(,@vars => ,ret)`, and where `,var` ≡ `,name` or `,name|,type`. When called, the method evaluates `,@body` (cf. Scheme's `LAMBDA`). The following a few example functions and their application:

```
((fun (x) x) 1) ==> 1
((fun (x|<int> => <int>) x) 2) ==> 2
((fun (x|...) x) 1 2 3) ==> (1 2 3)
((fun (x y|...) y) 1 2 3) ==> (2 3)
((fun (x => (tup <int>))) (tup x)) 1) ==> (tup 1)
```

The form `(DF ,name ,sig ,@body)` defines a function and binds it to the global variable `,name`.

A.2 Generics

Generic functions provide a form of polymorphism allowing many implementation methods with varying parameter types, called *specializers*. Methods on a given generic function are chosen according to applicability and are then ordered by specificity. A method is applicable if each argument is an instance of each corresponding specializer. A method A is more specific than method B if all of A's specializers are subtypes of B's. During method dispatch three cases can occur:

- if no methods are applicable then a no-applicable-method error is signaled,
- if methods are applicable but are not orderable then an ambiguous-method error is signaled,
- if methods are applicable and are orderable then the most specific method is called and the next methods are established.

The form `(DG ,name ,sig)` defines a global binding with name `,name` bound to a generic with signature `,sig`. The form `(DM ,name ,sig ,@body)` adds a method with signature `,sig` and body `,@body` to the generic `,name`, creating it if it does not exist. (cf., Dylan's `DEFINE METHOD`).

A.3 Classes and Types

Classes in GOO are by convention surrounded with angle brackets: `<` and `>`. The form `(DC ,name (,@parents))` creates a new class with the specified parents (note: GOO supports multiple inheritance).

Data fields on a class, called "properties" are added to a class after creation using the `DP` form. `(DP ,name (,oname|,owner => ,type) [,@init])` add's a property to class `,owner` with getter named `,name`, setter named `,name ## ''-setter''`, type `,type`, and optionally initial value `,init`.

All classes are first class objects and have a type `<class>`, which is a subclass of `<type>`. There are a variety of other types described in the main body of this paper which are all also subclasses of `<type>`.

New objects can be created from a `<class>` using the generic function `new (type|<type> prop-inits|...)`. This creates an instance of `type` with the specified properties.

Testing if a certain instance belongs to a type is accomplished via the `isa?` method; e.g. `(isa? 5 <num>) ==> #t`. Similarly, testing if a type is a subtype of another type is performed using the `subtype?` method; e.g. `(subtype? <int> <num>) ==> #t`.

A.4 Other data types

GOO has a variety of classes defined. Some of the important ones used in this document are:

`<any>` – all objects are derived from `<any>`

`<type>` – the abstract type of all types in GOO. Subclass of `<any>`.

`<class>` – a type which specifies an explicit inheritance relationship and can have instances created. Subclass of `<type>`.

`<mag>` – a magnitude. Totally orderable object. Subclass of `<any>`.

`<num>` – an abstract number. Subclass of `<mag>`.

`<int>` – an integer. Subclass of `<num>`.

`<flo>` – a floating point number. Subclass of `<num>`.

`<fun>` – an abstract function. Subclass of `<any>`.

`<met>` – a concrete method. Subclass of `<fun>`.

`<gen>` – a generic function. Dispatches to the most specific method. Subclass of `<fun>`.

`<col>` – an abstract collection of key/value pairs. Subclass of `<any>`.

`<seq>` – an abstract ordered collection of values. Subclass of `<col>`.

`<vec>` – a mutable constant-access-time sequence of values. Subclass of `<seq>`.

`<tup>` – an immutable constant-access-time sequence of values. Subclass of `<seq>`.

`<map>` – an abstract collection with explicit keys. Subclass of `<col>`.

`<tab>` – a hashtable. Subclass of `<map>`.

Appendix B

System Source Code

```
;; ================= A. =================
;; =========== type relations ===========
;; ======================================

(dc <typemod> (<type>))
  (dp type-val (<typemod> => <type>))
(dc <type-equal> (<typemod>))
(dc <type-sub> (<typemod>))
(dc <type-super> (<typemod>))

(df y= (x) (new <type-equal> type-val x))
(df y< (x) (new <type-sub> type-val x))
(df y> (x) (new <type-super> type-val x))
(dv o= t=)
(dv o< t<)

;;; EQTYPE?
(dg eqtype? (t1|<type> t2|<type> => <log>))
(dm eqtype? (t1|<type> t2|<type> => <log>)
  (and (subtype? t1 t2) (subtype? t2 t1)))


;;; ISTYPE?
(dg istype? (t1|<type> t2|<type> => <log>))

(dm istype? (t1|<type> t2|<type> => <log>)
  (eqtype? t1 t2))
(dm istype? (t1|<type> t2|<type-sub> => <log>)
  (subtype? t1 (type-val t2)))
(dm istype? (t1|<type-sub> t2|<type> => <log>)
  #f)
(dm istype? (t1|<type-sub> t2|<type-sub> => <log>)
  (subtype? (type-val t1) (type-val t2)))

(dm istype? (t1|<type> t2|<type-super> => <log>)
  (subtype? (type-val t2) t1))
(dm istype? (t1|<type-super> t2|<type> => <log>)
  #f)
(dm istype? (t1|<type-super> t2|<type-super> => <log>)
  (subtype? (type-val t2) (type-val t1)))

(dm istype? (t1|<type-super> t2|<type-sub> => <log>)
  (eqtype? (type-val t2) <any>))
(dm istype? (t1|<type-sub> t2|<type-super> => <log>)
  #f)

(dm istype? (t1|<type-equal> t2|<type> => <log>)
  (eqtype? (type-val t1) t2))
(dm istype? (t1|<type> t2|<type-equal> => <log>)
  (eqtype? t1 (type-val t2)))

(dm istype? (t1|<type-equal> t2|<type-equal> => <log>)
  (eqtype? (type-val t1) (type-val t2)))
(dm istype? (t1|<type-equal> t2|<type-sub> => <log>)
  (subtype? (type-val t1) (type-val t2)))
(dm istype? (t1|<type-sub> t2|<type-equal> => <log>)
  #f)

(dm istype? (t1|<type-equal> t2|<type-super> => <log>)
  (subtype? (type-val t2) (type-val t1)))
(dm istype? (t1|<type-super> t2|<type-equal> => <log>)
  #f)

(dm subtype? (t1|<type> t2|<typemod> => <log>)
  (istype? t1 t2))
(dm subtype? (t1|<union> t2|<typemod> => <log>)
  (istype? t1 t2))

(dm isa? (o t|<typemod> => <log>)
  (istype? (class-of o) t))


;; ================= B. =================
;; =========== type variables ===========
;; ======================================

;; <type-bot> is the type at the bottom of the type hierarchy.
;; It is a subtype of every type.
;; Parallels <any>, which every type is a subtype of.
;; can't use t= in the method signature for subtype?
;; so manually create a singleton.
(dc <type-bot-type> (<type>))
(dv <type-bot> (new <type-bot-type>))

(dm subtype? (t1|<type-bot-type> t2|<type> => <log>)
  #t)
(dm subtype? (t1|<type> t2|<type-bot-type> => <log>)
  #f)
(dm subtype? (t1|<type-bot-type> t2|<type-bot-type> => <log>)
  #t)

;====

;; the typevar type, and associated machinery.
(dv *typevars-pegged* #f)

(dv *typevar-states* (fab <tab> 5))

(df typevar-value-setter (z tv|<typevar>)
  (set (elt *typevar-states* tv) z)

(df typevar-value (tv|<typevar>)
  (elt-or *typevar-states* tv nul))

(dc <typevar> (<type>))
(dp name (x|<typevar> => <sym>) '?)

(dm == (o1|<typevar> o2|<typevar> => <log>) (= (name o1) (name o2)))
(dm id-hash (o|<typevar>)
  (id-hash (name o)))

(df typevar-min-type (x) (if (== x nul) <any> (1st x)))
(df typevar-max-type (x) (if (== x nul) <type-bot> (2nd x)))

;; special syntax to construct type variables
(ds (? ,name)
  `(new <typevar> name ',name))

;; FIXME: this code has to run in the evaluator because of a bug in the
;; compiler that keeps it from working otherwise..
(eval
(read-from-string "(seq

;; make order-mets use a pristene typevar state
(unless (bound? old-order-mets)
  (dv old-order-mets goo/boot:order-mets))

(df new-order-mets (m1|<met> m2|<met> args|<opts> => <sym>)
  (if (or (goo/boot:@fun-unification-vars m1) (goo/boot:@fun-unification-vars m2))
      (dlet ((*typevar-states* (fab <tab> 5))
             (*typevar-pegged* #t))
        (into *typevar-states* (goo/boot:@fun-unification-vars m1))
        (into *typevar-states* (goo/boot:@fun-unification-vars m2))
        (old-order-mets m1 m2 args))
      (old-order-mets m1 m2 args)))

(set goo/boot:order-mets new-order-mets)

;; same with met-app
(unless (bound? old-met-app?)
  (dv old-order-mets goo/boot:met-app?))

(df new-met-app? (met|<met> args|<opts> => <log>)
  (if (goo/boot:@fun-unification-vars met)
      (dlet ((*typevar-states* (fab <tab> 5)))
        (old-met-app? met args)
        (set (goo/boot:@fun-unification-vars met) *typevar-states*))
      (old-met-app? met args)))

(set goo/boot:met-app? new-met-app?)
)")
'goo/user)


;====

;; intersection of two types - returns the supertypes in common.
;; can either return a single type, or a type union.
;; e.g.
;; (type-intersection <int> <flo>) => <num>
;; (type-intersection <tup> <str>) => (t+ <flat> <seq.>)

(dg type-intersection (t1|<type> t2|<type> => <type>))

;; quite unoptimal, but works.
(dm type-intersection (t1|<class> t2|<class> => <type>)
  (let ((cl (rev (pick (fun (x) (mem? (class-ancestors t2) x))
                       (class-ancestors t1))))) ;remove rev when pick is fixed
    ;; find the minimal union all the common supertypes
    ;; assumes they are sorted from most specific to least specific
    ;; which is what class-ancestors conveniently returns
    (rep loop ((res '()) (in (enum cl)))
      (if (fin? in)
          (if (== (len res) 1)
              (1st res)
              (new <union> union-elts (rev res)))
          (let ((check (now in)))
            (if (any? (fun (x) (subtype? x check)) res)
                (loop res (nxt in))
                (loop (add res check) (nxt in))))))))

(dm type-intersection (t1|<type-su
;; FIXME: insert other type-intersection methods here...

(dm type-intersection (t1|<type-bot-type> t2|<type> => <type>)
  t2)
(dm type-intersection (t1|<type> t2|<type-bot-type> => <type>)
  t2)

;; union of two types - return a type encompassing both
;; can either return a single type, or a type union.
;; e.g.
;; (type-union <int> <flo>) => (t+ <int> <flo>)
;; (type-union <int> <num>) => <int>
;; should this be the default behavior for "t+"?

(dg type-union (t1|<type> t2|<type> => <type>))

(dm type-union (t1|<type> t2|<type> => <type>)
  (if (subtype? t1 t2)
      t1
      (if (subtype? t2 t1)
          t2
          (t+ t1 t2))))

;====

;; can t1 be a subtype of type variable t2?
;; if yes, adds a restriction to t2: must be supertype of t1.
;; ie set t2 to the intersection of t1 with t2

(dm subtype? (t1|<type> t2|<typevar> => <log>)
  (def val (typevar-value t2))
  ;; first make sure the new type is above the minimum of the type-var.
  (if (subtype? t1 (typevar-min-type val))
      (seq
        (unless *typevars-pegged*
          ;; now intersect the maximum of the typevar and the new type.
          (def new-type-max (type-intersection (typevar-max-type val) t1))
          (set (typevar-value t2) (tup
                                   (typevar-min-type val)
                                   new-type-max)))
        #t)
      #f))

;; can type variable t1 be a subtype of t2?
;; if yes, add restriction to t1: must be a subtype of t2.
(dm subtype? (t1|<typevar> t2|<type> => <log>)
  (def val (typevar-value t1))
  ;; first make sure the new type is below the maximum of the type-var.
  (if (subtype? (typevar-max-type val) t2)
      (seq
        (unless *typevars-pegged*
          ;; now union the minimum of the typevar and the new type.
          (def new-type-min (type-union (typevar-min-type val) t2))
          (set (typevar-value t1) (tup
                                   new-type-min
                                   (typevar-max-type val))))
        #t)
      #f))


;; we don't allow asking if one typevar is a subtype of another,
;; unless the typevars are pegged
;; doing so doesn't make it clear *which* type variable should be
;; side effected...
(dm subtype? (t1|<typevar> t2|<typevar> => <log>)
  (if *typevars-pegged*
      (subtype? (typevar-min-type (typevar-value t1))
                (typevar-min-type (typevar-value t2)))
      (error "subtype? comparison between type-variables is undefined.")))

(dm subtype? (t1|<union> t2|<typevar> => <log>)
  (def saved-typevar-states (dup *typevar-states*))
  (if (all? (fun (t) (subtype? t t2)) (union-elts t1))
      #t
      (seq
        (set *typevar-states* saved-typevar-states)
        #f)))

(dm subtype? (t1|<typevar> t2|<union> => <log>)
  (any? (fun (t) (subtype? t1 t)) (union-elts t2)))

(dm isa? (o t|<typevar> => <log>)
  (subtype? (class-of o) t))
```

```
;; ================ C. ================
;; ========== parametric types ==========
;; ====================================
(dv *class-param-table* (fab <tab> 10))

(dv *default-param-value* (gensym))

(dv *class-param-values* (fab <tab> 10))

(dv *class-param-defaults* (fab <tab> 10))

(dc <param> (<class>))

(df class-direct-params (x|<class> => <seq>)
  (elt-or *class-param-table* x '()))

(df class-direct-params-setter (z|<col> x|<class>)
  (set (elt *class-param-table* x) z))

(df class-params (x|<class> => <seq>)
  (del-dups
   (fold cat2 '() (map class-direct-params (rev (class-ancestors x))))))

(dv *empty-tab* (fab <tab> 0))
(df class-param-value-or (c|<class> k|<fun> default)
  (elt-or (elt-or *class-param-values* c *empty-tab*) k default))

(df class-param-value-setter (z c|<class> k|<fun>)
  (dv t (elt-or *class-param-values* c #f))
  (when (not t)
    (set t (fab <tab> 5))
    (set (elt *class-param-values* c) t))
  (set (elt t k) z))

(df class-param-value (c|<class> k|<fun>)
  (def v (class-param-value-or c k *default-param-value*))
  (if (== v *default-param-value*)
      (class-param-default k)
      v))

(df class-param-default (k|<fun>)
  (elt-or *class-param-defaults* k #f))

(df class-param-default-setter (z k|<fun>)
  (set (elt *class-param-defaults* k) z))
```

```
(ds (dparam ,name (,owner => ,type) ,default)
  `(seq
     (unless (bound? ,name)
         (dv ,name (goo/boot:fab-gen ',name '() (lst <any>) #f <any> '())))
     (def f (fun (x|(o< ,owner) => ,type) (class-param-value x ,name)))
     (set ,name (gen-add-met ,name f))
     (set (class-direct-params ,owner) (add (class-direct-params ,owner) ,name))
     (set (class-param-default ,name) ,default)
     )
  )

(ds (of ,name ,@args)
  (let ((newclassname (gensym)))
    `(seq
       (fun ()) ;; makes nasty hack work by forcing g2c on
       (dc ,newclassname (,name))
       ;; nasty hack! change the type of the new class from <class> to <param>
       (set (goo/boot:%object-class ,newclassname) <param>)
       ,(rep loop ((x (enum args)))
          (def key (now x))
          (def val (now (nxt x)))
          `(set (class-param-value ,newclassname ,key) ,val))
       ,newclassname)))


;; this form should not be necessary to use in the future
(ds (pdm ,name ,args ,code)
  `(seq
     (def f (fun ,args ,code))
     (unless (bound? ,name)
       (dv ,name (goo/boot:gen-from-met f)))

     (set (goo/boot:fun-unification-vars f) '(#t))
     (gen-add-met ,name f)))


(dm param-base (x|<param>)
  (param-base (1st (class-parents x))))

(dm param-base (x|<any>)
  x)

(dm param-compat? (x1|<type> x2|<type>)
  (istype? x1 x2))

(dm param-compat? (x1 x2)
  (== x1 x2))
```

```
(df param-subtype? (t1 t2)
  (or (@subclass t1 t2)
      (and (subtype? (param-base t1) (param-base t2))
           (all? (fun (f)
                   (param-compat? (f t1) (f t2))
                 (class-params t2)))))

(dm subtype? (t1|<param> t2|<class> => <log>)
  (param-subtype t1 t2 ))

(dm subtype? (t1|<class> t2|<param> => <log>)
  (param-subtype t1 t2))

(dm subtype? (t1|<param> t2|<param> => <log>)
  (param-subtype t1 t2))

(dm isa? (o t|<param> => <log>)
  (subtype? (class-of o) t))
```

Bibliography

[Bac02] Jonathan Bachrach. *GOO Reference Manual*, 2002. 9

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998. 12

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *Lecture Notes in Computer Science*, 1445:523–??, 1998.

[CCH$^+$89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, pages 273–280. ACM Press, 1989.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, New York, NY, 1997.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. Massachusetts Institute of Technology, August 1990.

[Pau91] Laurence C. Paulson. *ML for the working programmer*. Cambridge University Press, 1991. 12

[Sha96] A. Shalit. *The Dylan Reference Manual*. Addison Wesley, 1996. 11, 51

[Str91] Bjarne Stroustrup. *The C++ programming language (2nd ed.)*. Addison-Wesley Longman Publishing Co., Inc., 1991. 11

[TT99] Kresten Krab Thorup and Mads Torgersen. Unifying genericity: Combining the benefits of virtual types and parameterized classes. *Lecture Notes in Computer Science*, 1628:186–??, 1999. 13