Adding Wildcards to the Java Programming Language

Mads Torgersen
Christian Plesner Hansen
Erik Ernst, and
Peter von der Ahé
University of Aarhus
Åbogade 34
DK-8200 Århus N, Denmark
{madst,plesner,eernst,pahe}@daimi.au.dk

Gilad Bracha and
Neal Gafter
Sun Microsystems, Inc.
4150 Network Cycle
Santa Clara, CA 95054, USA
{Gilad.Bracha,Neal.Gafter}@sun.com

ABSTRACT

This paper describes *wildcards*, a new language construct designed to increase the flexibility of object-oriented type systems with parameterized classes. Based on the notion of use-site variance, wildcards provide a type safe abstraction over different instantiations of parameterized classes, by using '?' to denote unspecified type arguments. Thus they essentially unify the distinct families of classes often introduced by parametric polymorphism. Wildcards are implemented as part of the upcoming addition of generics to the Java™ programming language, and will thus be deployed world-wide as part of the reference implementation of the Java compiler `javac` available from Sun Microsystems, Inc. By providing a richer type system, wildcards allow for an improved type inference scheme for polymorphic method calls. Moreover, by means of a novel notion of *wildcard capture*, polymorphic methods can be used to give symbolic names to unspecified types, in a manner similar to the "`open`" construct known from existential types. Wildcards show up in numerous places in the Java Platform APIs of the upcoming release, and some of the examples in this paper are taken from these APIs.

Categories and Subject Descriptors

D.3.3 [**Language Constructs and Features**]: Classes and objects, Data types and structures, Polymorphism

Keywords

Wildcards, genericity, parameterized types

1. INTRODUCTION

Parametric polymorphism is well-known from functional languages such as Standard ML [22], and over the past two decades similar features have been added to a number of object-oriented languages [21, 28, 13].

For some time it has been clear that the Java programming language was going to be extended with parametric polymorphism in the form of *parameterized classes* and *polymorphic methods*, i.e., classes and methods with type parameters. A similar mechanism has recently been described for C# [10], and is likely to become part of a future version of that language [18].

The decision to include parametric polymorphism – also known as *genericity* or *generics* – in the Java programming language was preceded by a long academic debate. Several proposals such as GJ and others [25, 1, 24, 4, 7] were presented, thus advancing the field of programming language research. It became increasingly clear that the mechanism on its own, imported as it were from a functional context, lacked some of the flexibility associated with object-oriented subtype polymorphism.

A number of proposals have sought to minimize these problems [8, 9, 2, 5, 6], and an approach by Thorup and Torgersen [30], which we shall refer to as *use-site variance*, seems particularly successful in mediating between the two types of polymorphism without imposing penalties on other parts of the language. The approach was later developed, formalized, and proven type sound by Igarashi and Viroli [17] within the Featherweight GJ calculus [15]. This work addresses typing issues, but was never implemented full-scale.

Wildcards are the result of a joint project between The University of Aarhus and Sun Microsystems, Inc., in which we set out to investigate if these theoretical proposals could be adapted and matured to fit naturally into a future version of the language extended with parametric polymorphism, and whether an efficient implementation was feasible.

The project has been very successful in both regards. The core language mechanism has been reworked syntactically and semantically into *wildcards* with a unified and suggestive syntax. The construct has been fully integrated with other language features – particularly polymorhic methods – and with the Java platform APIs, leading to enhanced

SAC'04, March 14-17, 2004, Nicosia, Cyprus

expressiveness, simpler interfaces, and more flexible typing. The implementation within the Java compiler is an extension of the existing generics implementation, enhancing the type checker and erasing parametric information to produce type-safe non-generic bytecode. Our implementation of wildcards and the associated modifications are now scheduled to be part of the forthcoming release of the Java platform (JDK1.5).

The development process has raised a wealth of interesting theoretical and implementational issues. The focus of this paper, however, is on what is probably most important to users of the language: the new language constructs, and the problems they address. While our experiences are specific to the Java programming language, wildcards should be equally well suited for other object-oriented languages, such as C#, having or planning an implementation of parametric polymorphism.

In the following, we will describe wildcards relative to GJ [4], a proposed dialect of the Java programming language with generics, which is the starting point for the effort of introducing genericity in the Java platform. We are thus assuming a language with parametric polymorphism and describing wildcards as an extension of this language, although there will never in reality be a release of the Java platform with generics but without wildcards.

The central idea of wildcards is pretty simple. Generics in the Java programming language allow classes like the Java platform API class List to be parameterized with different element types, e.g., List⟨Integer⟩ and List⟨String⟩. In GJ there is no general way to abstract over such different kinds of lists to exploit their common properties, although polymorphic methods may play this role in specific situations. A wildcard is a special type argument '?' ranging over all possible specific type arguments, so that List⟨?⟩ is the type of all lists, regardless of their element type.

The contributions described in this paper include:

- The wildcard mechanism in itself, which syntactically unifies and semantically generalizes the set of constructs constituting use-site variance
- An enhanced type inference scheme for polymorphic methods, exploiting the improved possibilities for abstraction provided by wildcards
- A mechanism which we call *wildcard capture*, that in some type safe situations allow polymorphic methods to be called even though their type arguments cannot be inferred

An additional contribution is the implementation itself, whose existence and industrial-quality standard is a proof of the possibility and practicality of wildcard typing in a real setting. The current prototype can be downloaded for inspection and evaluation at `http://developer.java.sun.com/developer/earlyAccess/adding_generics/index.html`.

In Section 2 we introduce the wildcard construct itself, describing how it can be used and why it is typesafe. Section 3 investigates the integration with polymorphic methods that leads to improved type inference and wildcard capture. Related work is explored in Section 4, and Section 5 concludes.

2. TYPING WITH WILDCARDS

The motivation behind wildcards is to increase the flexibility of generic types by abstracting over the actual arguments of a parameterized type. Syntactically, a wildcard is an expression of the form '?', possibly annotated with a bound, as in '? **extends** T' and '? **super** T', where T is a type. In the following we describe the typing of wildcards, and the effect of using bounds.

2.1 Basic Wildcards

Prior to the introduction of generics into the Java programming language, an object of type List was just a list, not a list of any specific type of object. However, often all elements inserted into a list would have a common type, and elements extracted from the list would be viewed under that type by a dynamic cast. To make this usage type safe, GJ lets classes like List be parameterized with an element type. Objects inserted must then have that type, and in return extracted objects are known to have that type, avoiding the unsafe cast. In most cases, this is an improvement over the previous non-generic scheme, but it makes it harder to treat a list as "just a list", independent of the element type. For instance, a method could take a List as an argument and only be interested in clearing it or reading properties like the length. In GJ, that could be expressed using a polymorphic method with a dummy type variable:

```
⟨T⟩ void aMethod(List⟨T⟩ list) { ... }
```

The solution is to give a name to the actual element type of the list and then ignore it in the body of the method. This is not a clean solution—but it works and was used extensively in GJ's libraries.

A more serious problem is the case where a class needs a *field* whose type is some List, independent of the element type. This is especially a problem in cases where the generic class provides a lot of functionality independent of the actual type parameters, as is the case for instance with the generic version of the class java.lang.Class. This cannot be expressed in GJ.

The solution is to use an *unbounded wildcard*, '?', in place of the type parameter when the actual type is irrelevant:

```
void aMethod(List⟨?⟩ list) { ... }
```

This expresses that the method argument is some type of list whose element type is irrelevant. Similarly, a field can be declared to be a List of anything:

```
private List⟨?⟩ list;
```

The type List⟨?⟩ is a supertype of List⟨T⟩ for any T, which means that any type of list can be assigned into the list field. Moreover, since we do not know the actual element type we cannot put objects into the list. However, we are allowed to read Objects from it—even though we do not know the exact type of the elements, we do know that they will be Objects.

In general, if the generic class C is declared as

```
class C⟨T extends B⟩ { ... }
```

when called on a C⟨?⟩, methods that return T will return the declared bound of T, namely *B*, whereas a method that expects an argument of type T can only be called with null. This mean that we *can* actually add elements to a List⟨?⟩, but only nulls.

Note that a wildcard should not in general be considered as a name of a specific type. For instance, the two occurrences of '?' in Pair⟨?,?⟩ are not assumed to stand for the same type, and even for the list shown above, the '?' in its type may stand for two different types before and after an assignment, as in list = new List⟨String⟩(); list = new List⟨Integer⟩().

2.2 Bounded Wildcards

Unbounded wildcards solve a class of problems with generic types, but lack one capability of polymorphic methods: if the element type of a list is not completely irrelevant, but required to conform to some bound, this could be expressed in GJ using a type bound (here Number):

```
⟨T extends Number⟩ void aMethod(List⟨T⟩ list) { ... }
```

To be able to express that the element type of the list must be a subtype of Number, we again have to introduce a dummy type variable. As before, this only works for methods and not for fields. In order for wildcards to help us out once more, we therefore equip them with *bounds* to express the range of possible type arguments "covered" by the wildcard:

```
void aMethod(List⟨? extends Number⟩ list) { ... }
```

This expresses that the method can be called with any list, as long as the element type is a subtype of Number. As before, we cannot write (anything but null) to the list, since the actual element type is unknown, but we are now allowed to read Numbers from it:

```
List⟨? extends Number⟩ list = new ArrayList⟨Integer⟩();
Number num = list.get(0); // Allowed
list.set(0, new Double(0.0)); // Illegal!
```

Parameterized types with extends-bounded wildcards are related by subtyping in a *covariant* fashion: All instances of List⟨? extends Integer⟩ are also instances of List⟨? extends Number⟩, so the former is a subtype of the latter.

While extends-bounds introduce *upper* bounds on wildcards, the introduction of *lower* bounds is also possible using so-called super-bounds. The type List⟨? super String⟩ is a supertype of any List⟨*T*⟩ where *T* is a supertype of String; for instance List⟨String⟩ and List⟨Object⟩.

This is useful, e.g., with Comparator objects. The Java platform class TreeSet represents a tree of elements that are ordered. One way to define the ordering is to construct the TreeSet with a specific Comparator object:

```
interface Comparator⟨T⟩ {
  int compareTo(T fst, T snd);
}
```

When constructing, e.g., a TreeSet⟨String⟩, we need to pass it some Comparator that can compare Strings. This can be done by a Comparator⟨String⟩, but a Comparator⟨Object⟩ will do just as well, since Strings are Objects. In this case, the type Comparator⟨? super String⟩ is appropriate, since it is a supertype of any Comparator⟨*T*⟩ where *T* is a supertype of String.

Conversely to extends-bounds, super-bounds give rise to *contravariant* subtyping: Comparator⟨? super Number⟩ is a subtype of Comparator⟨? super Integer⟩.

3. POLYMORPHIC METHODS

Wildcards and polymorphic methods interact in several ways, as described in the following.

3.1 Type Inference

Polymorphic methods can be called with or without explicit type arguments. When no explicit type arguments are given, they are inferred from the type information available at the call site. Inferring a type for a type variable T means selecting a type that by insertion produces a method signature such that the given call site is type correct, and ensuring that this type satisfies the bound for T. In this process a subtype is preferred over a supertype because the former generally preserves more information about return values. To be concrete, consider these declarations:

```
⟨T⟩ T choose(T a, T b) { ... }

Set⟨Integer⟩ intSet = ...
List⟨String⟩ stringList = ...
```

In the call choose(intSet, stringList), a type has to be found for T that is a supertype of both Set⟨Integer⟩ and List⟨String⟩. Since different parameterizations of the same class are incomparable in GJ the only such type is Object, even though Set⟨T⟩ and List⟨T⟩ share the superinterface Collection⟨T⟩. What lacks is the ability to describe a Collection whose element type is not specified directly, but abstracts over both Integer and String. With wildcards, this can be expressed as Collection⟨?⟩, and hence a more specific type than Object can be inferred.

This is an example of a general phenomenon: when given two parameterized classes with different type arguments for the same parameter, it is inherently impossible for GJ to infer a type that involves that parameter. In this case that means ignoring that Collection⟨T⟩ is a common superinterface for Set⟨T⟩ and List⟨T⟩. This restriction does not apply when wildcards are available, because ? can be used in any case, and that leads to a more accurate type inference.

In the choose() example, the type variable T is also used as a return type, so the improved inference has the beneficial consequence that the caller now knows that a Collection is returned—useful if the intention is, e.g., to iterate over the elements.

In some cases, the inference may be improved to provide bounds for the inferred wildcards. Our experiments show, however, that a general approach to obtain the best possible

bounds has some problems. First, there may be both an upper and a lower "best bound", so the choice between them would have to be arbitrary. Secondly, the best upper or lower bound may be an infinite type, with all the problems that this entails. In our current implementation we take instead a simplistic strategy, allowing bounds in the inference result only if they occur in one of the type arguments on which the inference is based, *and* are implied by the other. Thus, for Set⟨Integer⟩ and List⟨? **extends** Number⟩, we would infer Collection⟨? **extends** Number⟩.

3.2 Wildcard Capture

Wildcards, as described above, turn out to cause a practical dilemma when defining methods and variables. An example of this is the static Collections.unmodifiableSet() method, which constructs a read-only view of a given set. A natural signature for this method could be this:

⟨T⟩ Set⟨T⟩ unmodifiableSet(Set⟨T⟩ set) { ... }

This method can be called with a Set⟨*T*⟩ for any type *T*, and returns a set with the same element type. However, it cannot be called with a Set⟨?⟩, because the actual element type is unknown. A read-only view of a set is useful even if the actual element type is unknown, so this is a problem. However, since the body of this method does not depend on the exact element type, it could instead be defined as

Set⟨?⟩ unmodifiableSet(Set⟨?⟩ set) { ... }

This would allow the method to accept any set, but in return discards the information that the returned set has the same element type as the given set:

```
Set⟨String⟩ set = ...
Set⟨String⟩ readOnly = unmodifiableSet(set); // Error!
```

In this case we get an error because the result of calling unmodifiableSet with a Set⟨Integer⟩ is a Set⟨?⟩. And so, we are left with a choice: should the method take a Set⟨T⟩ to give an accurate return type or a Set⟨?⟩ to allow the method to be called with sets whose exact element type is unknown?

The solution is to observe that it is actually safe to allow the method taking a Set⟨T⟩ to be called with a Set⟨?⟩. We may not know the actual element type of the Set⟨?⟩, but we know that at the instant when the method is called, the given set will have some specific element type, and *any* such element type would make the invocation typesafe. This mechanism of allowing a type variable to be instantiated to a wildcard in some situations is known as *wildcard capture*, because the actual run-time type behind the ? is "captured" as T in the method invocation.

Capturing wildcards is only legal in some situations, however. It must be known that there is a unique type to capture at runtime. For this reason, a type variable can only capture one wildcard because, for instance, the actual element types of two different Set⟨?⟩s may be different. Also, only type variables that occur at "top level" in a generic class (as in Stack⟨T⟩ and unlike Stack⟨Stack⟨T⟩⟩ or Stack⟨T[]⟩), can be captured. This is again because two Stack⟨?⟩ elements of a Stack⟨Stack⟨?⟩⟩ may have different element types, and so cannot be captured by the single T in Stack⟨Stack⟨T⟩⟩.

The first definition of unmodifiableSet() above fulfills these conditions, so the following call is allowed:

```
Set⟨?⟩ set = ...;
set = unmodifiableSet(set);
```

Thus, the API needs to contain only the polymorphic version of unmodifiableSet() since, with capture, it implies the typing also of the wildcard version.

3.3 Proper Abstraction

Wildcard capture also addresses a related problem of GJ. Consider the method Collections.shuffle(), which takes a List and reshuffles its elements. One possible choice of signature is

⟨T⟩ **void** shuffle(List⟨T⟩ list) { ... }

because the method body needs a name for the element type of the list, in order to remove and re-insert its elements. However, the caller of such a method should only have to worry about the types of objects the method can be called with; in this case any List. Seen from the caller's perspective the signature of shuffle() should therefore be the more concise:

void shuffle(List⟨?⟩ list) { ... }

Wildcard capture allows us to mediate between these two needs, because it makes it possible for the wildcard version of the method (which should be public) to call the polymorphic version (which should be private and have a different name).

In general, private methods can be employed in this way to "open up" the type arguments of types with wildcards, thus avoiding that implementation details such as the need for explicit type arguments influence the public signatures of a class or interface declaration.

3.4 Capture and Quantification

Wildcard capture further exploits the connection between wildcards and the existentially quantified types of Mitchell and Plotkin [23], which is established for variant parametric types in [17]. Following this line of argument, the above declaration of Set⟨?⟩ set can be compared to a similar declaration with the existential type ∃X.Set⟨X⟩.

Capture then amounts to applying the **open** operation of existential types to obtain a name T for the particular element type of set *and* a name s for set with the type Set⟨T⟩. Both can then be used in a subexpression containing the method call to be captured:

```
∃X.Set⟨X⟩ set;
open set as T,s in Collections.unmodifiableSet(s);
```

Using this syntax it is clear that unmodifiableSet() is in fact called with a fixed type argument T, because s has the type Set⟨T⟩. Wildcard capture may therefore be seen as an implicit wrapping of polymorphic method calls with such **open** statements, when appropriate.

4. RELATED WORK

Virtual types are the ultimate origins of wildcards, and the historical and semantic relations are described below. We then look at variance annotations both at the declaration site and the use site of parametric classes, the latter approach being the starting point for the design of the wildcard mechanism. Finally, we investigate the origins of the expressive differences between polymorphic methods and wildcards which we saw in Section 2.

4.1 Virtual types

Wildcards ultimately trace their origins back to the BETA programming language [20]. Virtual classes in BETA support genericity, thereby providing an alternative to parameterized classes. Virtual classes are members of classes that can be redefined in subclasses, similarly to (virtual) methods. In their original form in BETA, virtual classes were a happy by-product of BETA's unification of methods and classes into *patterns*, and so the mechanism in BETA is actually known as *virtual patterns*. Thorup introduced the term *virtual type* in his proposal for adding these to the Java programming language [29]. This terminology was followed by subsequent incarnations of the construct [31, 6, 16], which all re-separate virtual types from virtual methods.

Using Thorup's syntax, a generic List class may be declared as follows:

```
abstract class List {
  abstract typedef T;
  void add(T element) { ... }
  T get(int i) { ... }
}
```

T is a type attribute, which may be further specified in subclasses. These can either *further bound* the virtual type by constraining the choice of types for T, or they can *final bind* it by specifying a particular type for T:

```
abstract class NumberList {
  abstract typedef T as Number; // Further bounding
}

class IntegerList extends NumberList {
  final typedef T as Integer; // Final binding
}
```

These classes are arranged in a subtype hierarchy,

$$\mathsf{IntegerList} <: \mathsf{NumberList} <: \mathsf{List},$$

which is very similar to that of a parameterized List class with wildcards:

$$\mathsf{List}\langle\mathsf{Integer}\rangle <: \mathsf{List}\langle?\ \mathbf{extends}\ \mathsf{Number}\rangle <: \mathsf{List}\langle?\rangle$$

Also, the abstract List classes—those with non-final virtual types—restrain the use of their methods, so that an attempt to add e.g. an Integer to a NumberList will be rejected.[1]

[1]Actually in BETA, assignments that may possibly succeed are not rejected by the compiler: instead a warning is issued and a runtime cast is automatically inserted. This policy has lead many to the false conclusion that BETA and virtual types are not statically safe; see, e.g., [6, 31].

Thus, virtual types in BETA is the first mechanism that lets different parameterizations of a generic class share a nontrivial common supertype. However, since subtypes are always subclasses, achieving hierarchies like the above requires planning: if IntegerList had been a direct subclass of List, it could not also be a subtype of NumberList. Furthermore, the use of single inheritance prohibits multiple supertypes, whereas wildcards allow, e.g., List⟨Integer⟩ to be a subtype of both Collection⟨Integer⟩ and List⟨?⟩.

The gbeta language [14], which generalizes BETA in several ways, reduces the latter problem by having structural subtyping at the level of mixins, but the inheritance hierarchy must still be carefully planned and centrally managed. However, inspired by various variance mechanisms including [30] the notion of *constrained virtuals* has recently been added to gbeta, thus providing a purely structural mechanism integrated with virtual patterns.

Thorup and Torgersen [30] compare the two genericity mechanisms, parameterized classes and virtual types, seeking to enhance each with the desirable features of the other. Virtual types are thus extended with the structural subtyping characteristic of parameterized classes (relating List⟨Number⟩ to Collection⟨Number⟩) to overcome the restrictions of BETA above. This approach has later been used in the Rune project [32].

4.2 Declaration-site variance

A different approach to obtain subtyping relationships among different instantiations of parameterized classes is to use *variance annotations*. First proposed by Pierre America [2], and later used in the Strongtalk type system [3], declaration-site variance allows the declaration of type variables in a parameterized class to be designated as either co- or contravariant. For instance, a read-only (functional) List class may be declared as:

```
class List⟨covar T⟩ {
  T head() { ... }
  List⟨T⟩ tail() { ... }
}
```

This will have the effect that, e.g., List⟨Integer⟩ is a subtype of List⟨Number⟩, but prevents the List class from having methods using T as the type of an argument. In a symmetric fashion, write-only structures, such as output streams, can be declared contravariant in their type arguments.

In practical object-oriented programming, this approach has severe limitations. Usually, data structures such as collections have both read and write operations using the element type, and in that situation, declaration-site variance cannot be applied.

Note that "write operation" is to be taken in the broad sense of "operations taking arguments of the element type". Thus, due to the covariance annotation the above functional List class cannot even contain a cons() method of the following form:

```
List⟨T⟩ cons(T elm) { return new List⟨T⟩(elm,this); }
```

even though this does not modify the list. Thus, in reality, declaration-site variance enforces a functional or procedural style of programming, where a lot of functionality has to be placed outside of the classes involved.

4.3 Use-site Variance

Thorup and Torgersen introduce the concept of *use-site covariance* for parameterized classes [30]. This is a new way of providing *covariant* arguments to parameterized classes, inspired by BETA. A prefix '+' is used, and List⟨+Number⟩ denotes a common supertype of all List⟨T⟩, where T is a subtype of Number. In exchange for the covariance, writing to a List⟨+Number⟩ is prohibited. Hence, '+Number' is essentially equivalent to the wildcard '? **extends** Number'.

In [17], Igarashi and Viroli propose a significant extension of this scheme, adding a *contravariant* form of use-site variance List⟨-Number⟩, roughly equivalent to List⟨? **super** Number⟩. Also, a so-called "bivariant" form List⟨*Number⟩ is added, which, like an unbounded wildcard List⟨?⟩ ranges over all kinds of lists. In the bivariant case, the Number part of the type argument is ignored and is there only for syntactic symmetry. The authors themselves propose the shorthand List⟨*⟩. Igarashi and Viroli provide a formalization in context of Featherweight GJ [15], which has been proven sound. Their work was our starting point for the design of wildcards, and the differences between this approach and the approaches we know from languages like BETA has been a source of fruitful discussions. For instance, we currently use covariance propagation as described by Igarashi and Viroli, but this does not fully describe our approach because it does not cover wildcard capture. A formalization that covers all the features is ongoing work.

Unlike wildcards, co- and contravariant instantiations of use-site variance rely heavily on read-only and write-only semantics. With the contravariant List⟨-Number⟩, for instance, calling the get() method is strictly disallowed, because the class is considered write-only. Conversely, calling add() on a covariant List⟨+Number⟩ is prohibited, even with null, and of course the bivariant List⟨*Number⟩ disallows both.

We find the focus on read-only and write-only somewhat misleading, especially because it seems to imply a kind of protection. For instance, a programmer might well consider a List⟨+Number⟩ to be safe-guarded from mutation, but in reality it is still perfectly possible to call e.g. its clear() method, because it does not take arguments of the element type.

Wildcards focus instead on the type information trade-off: The less you require in a type, the more objects can be typed by that type. For example, the type List⟨? **super** Integer⟩ describes a larger set of objects than the type List⟨Integer⟩; in both cases it is harmless to call a read method like get(), but in the latter case we know the result is an Integer and in the former case we only know it is an Object.

4.4 Polymorphic methods

Parametric polymorphism, as known from SML and provided for object-orientation through languages such as Pizza and GJ, provides insufficient support for abstracting over different instantiations of a parameterized class. The expressive power added by wildcards and similar mechanisms will be substantiated in the following.

Parametric polymorphism in functional languages is a highly effective mechanism for abstracting over different instantiations of a polymorphic type. For instance, the polymorphic type of the well-known map function,

$$(\alpha \to \beta) \to \alpha \text{ list} \to \beta \text{ list},$$

allows us to use map on *any* list and function where the argument type of the function is the element type of the list. This seems to indicate that there is no need for additional mechanisms in order to abstract over different instantiations of a parameterized class.

However, the abstraction over the values of type arguments such as α and β is handled by statically ensuring that every function invocation represents a consistent (partial) instantiation of the polymorphic type. This means that map can only be invoked if we statically know the value of each type variable, or it is expressed in terms of variables in the type of the calling function, etc., down to the top-level expression associated with the bottom of the call stack. The type of a top-level expression cannot contain type variables, according to the value restriction rule of SML'97 [19], unless it is syntactically a simple value (in which case the computation and hence also the run-time stack is trivial). Consequently, in order for a polymorphic method to be invoked, full type information must be available somewhere in the program.

This is no problem in purely functional languages, because each value itself may be derived from the call stack and hence it does not represent a loss of flexibility that its type must be deducible from the stack. In a language with side-effects, however, a value can be stored by one part of the program and later read by another part. Even if full type information were available about the value when it was stored, some of that information may be lost at the point where the value is read.

In an object-oriented context the same phenomenon arises for invocations of polymorphic methods, but here mutable data structures are the norm rather than the exception. Compared to the typical programming assumptions of object-oriented languages, requiring that all type information about the type arguments to parameterized classes is available somewhere in the program, as in GJ or generic C#, is a serious restriction. If we create, e.g., an Integer object and store it in a variable of type Number then there is no need for *any* location in the program where the exact type Integer of that object is known. The need to maintain exact type knowledge for individual objects such as a List⟨Integer⟩ runs contrary to the object-oriented polymorphic style of programming, and is a heavy burden on programmers.

In other words, when directly carrying over polymorphic methods from functional languages into object-oriented languages, the need to maintain exact type information leads to more complex and contorted designs, especially in large scale systems. Wildcards alleviate this problem, and are hence a significant enhancement of the expressive power of the language.

One way to state this result is that existential types do make a difference, and wildcards provide a restricted kind of existential types. Note that there are also several proposals for extending SML with existential types, e.g. [27] where structures and functors are made first-class and typed existentially, and such extensions seems to be a viable strategy to remove the value restriction from SML.

5. CONCLUSIONS

In this project, the Java programming language has been extended with wildcards, thus bringing ideas about virtual types and use-site variance to the mainstream. In this design and implementation process, several lessons were learned and new ideas produced. First, the notion of wildcards was designed and implemented; second, type inference for invocation of polymorphic methods was enhanced to handle wildcards; and third, the notion of wildcard capture was introduced, exploiting the existential nature of the '?' in many usages of wildcards. Finally, we have argued that the expressive power of wildcards is a non-trivial enhancement to the power of polymorphic methods, essentially because wildcards are a restricted form of existential types.

6. ACKNOWLEDGMENTS

Numerous people have contributed to the design of wildcards, not least from the participation in many lively discussions. We wish here to thank especially Martin Odersky, Atsushi Igarashi, Mirko Viroli, Lars Bak, Josh Bloch and Graham Hamilton who all had a significant influence on the resulting design.

7. REFERENCES

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the java programming language. In *Object Oriented Programming: Systems, Languages and Applications*, Atlanta, Georgia, Oct. 1997. OOPSLA97, ACM Press. Toby Bloom, editor.

[2] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Object Oriented Programming: Systems, Languages and Applications/European Conference on Object-Oriented Programming*, pages 161–168, Ottawa, Canada, Oct. 1990. OOPSLA/ECOOP90, ACM Press. Norman K. Meyrowitz, editor.

[3] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *Object Oriented Programming: Systems, Languages and Applications*, Washington DC, Oct. 1993. OOPSLA93, ACM Press. Andreas Paepcke, editor.

[4] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the java programming language. In OOPSLA98 [26].

[5] K. Bruce. Subtyping is not a good match for object-oriented programming languages. In ECOOP97 [11].

[6] K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *European Conference on Object-Oriented Programming*, Brussels, Belgium, July 1998. ECOOP98, LNCS 1445, Springer Verlag. Eric Jul, editor.

[7] R. Cartwright and G. L. Steele. Compatible genericity with runtime-types for the Java programming language. In OOPSLA98 [26].

[8] W. Cook. A proposal for making Eiffel type-safe. In *European Conference on Object-Oriented Programming*, pages 57–70, Nottingham, England, July 1989. ECOOP89, Cambridge University Press. Stephen Cook, editor.

[9] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Principles of Programming Languages*, pages 125–135, San Francisco, California, Jan. 1990. POPL90, ACM Press. Paul Hudak, editor.

[10] ECMA. C# language specification. `http://www.ecma-international.org/publications/standards/Ecma-334.htm`, 2002.

[11] ECOOP97. *European Conference on Object-Oriented Programming*, Jyväskylä, Finland, June 1997. LNCS 1241, Springer Verlag. Mehmet Akşit and Satoshi Matsuoka, editors.

[12] ECOOP99. *European Conference on Object-Oriented Programming*, Lisbon, Portugal, June 1999. LNCS 1628, Springer Verlag. Rachid Guerraoui, editor.

[13] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[14] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[15] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages and Applications*, pages 132–146, Denver, Colorado, Oct. 1999. OOPSLA99, ACM Press. Linda Northrop, editor.

[16] A. Igarashi and B. C. Pierce. Foundations for virtual types. In ECOOP99 [12].

[17] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *European Conference on Object-Oriented Programming*, pages 441–469, Málaga, Spain, June 2002. ECOOP02, LNCS 2374, Springer Verlag. Boris Magnusson, editor.

[18] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 1–12, N.Y., June 20–22 2001. ACMPress.

[19] D. MacQueen. SML '97 conversion guide. `http://www.smlnj.org/doc/Conversion/index.html`, 2003.

[20] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[21] B. Meyer. Genericity versus inheritance. In *Object Oriented Programming: Systems, Languages and Applications*, pages 391–405, Portland, Oregon, Nov. 1986. OOPSLA86, ACM Press. Norman K. Meyrowitz, editor.

[22] R. Milner, M. Tofte, R. W. Harper, and D. MacQueen. *The Definition of Standard ML*. MIT Press, 1997.

[23] J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 1988.

[24] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *Conf. Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Paris, France, Jan. 1997. POPL97, ACM Press. Neil D. Jones, editor.

[25] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 146–159, Paris, France, 15–17 Jan. 1997.

[26] OOPSLA98. *Object Oriented Programming: Systems, Languages and Applications*, Vancouver, BC, Oct. 1998. ACM Press. Craig Chambers, editor.

[27] C. V. Russo. First-class structures for standard ML. *Lecture Notes in Computer Science*, 1782:336++, 2000.

[28] D. Stoutamire and S. Omohundro. The Sather 1.1 specification. Technical Report TR-96-012, International Computer Science Institute, Berkeley, CA, Aug. 1996.

[29] K. K. Thorup. Genericity in Java with virtual types. In ECOOP97 [11], pages 444–471.

[30] K. K. Thorup and M. Torgersen. Unifying genericity. In ECOOP99 [12], pages 186–204.

[31] M. Torgersen. Virtual types are statically safe. In K. Bruce, editor, *5th Workshop on Foundations of Object-Oriented Languages*, San Diego, CA, Jan. 1998.

[32] M. Torgersen. *Unifying Abstractions*. PhD thesis, Computer Science Department, University of Aarhus, bogade 34, DK-8200 rhus N, Sept. 2001.