JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2004

Vol. 3, No. 11
Special issue: OOPS track at SAC 2004, Nicosia/Cyprus

Adding Wildcards to the Java Programming Language

Mads Torgersen, University of Aarhus, Denmark
Erik Ernst, University of Aarhus, Denmark
Christian Plesner Hansen, OOVM, Aarhus, Denmark
Peter von der Ahé, Sun Microsystems, Inc., California, USA
Gilad Bracha, Sun Microsystems, Inc., California, USA
Neal Gafter, Google Inc., California, USA

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 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 that parametric polymorphism introduces. Wildcards are implemented as part of the addition of generics to the Java™ programming language, and is thus 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 newest release, and some of the examples in this paper are taken from these APIs.

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, 12].

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# [11], and is likely to become part of a future version of that language [19].

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, 8] were presented, thus advancing the field of programming language research. It became increasingly

Cite this article as follows: Mads Torgersen, Erik Ernst, Christian Plesner Hansen, Peter von der Ahé, Gilad Bracha, Neal Gafter: "Adding Wildcards to the Java Programming Language", in *Journal of Object Technology*, vol. 3, no. 11, December 2004, Special issue: OOPS track at SAC 2004, Nicosia/Cyprus, pp. 97–116, http://www.jot.fm/issues/issue_2004_12/article5

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 [9, 10, 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 [18] within the Featherweight GJ calculus [16]. This work addresses typing issues, but was never implemented full-scale.

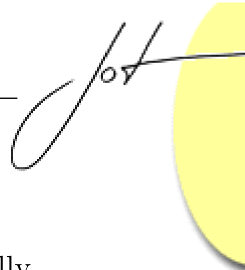
Wildcards are the result of a joint project between the University of Aarhus and Sun Microsystems, Inc., in which we set out to investigate if these theoretical proposals could be adapted and matured to fit naturally into the language extended with parametric polymorphism, and whether an efficient implementation 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 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 part of the recent release of the Java 2 Standard Edition Development Kit version 5.0 (J2SE 5.0).

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 was 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.

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. Section 4 briefly describes how we implemented wildcard capture. Related work is explored in Section 5, and Section 6 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.

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 old, 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 only to clear it or to read 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⟩ { ... }
```

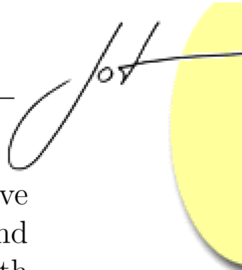
when called on a C⟨?⟩, methods that return T will return the declared bound of T, namely *B*, whereas a method that expects an argument of type T can only be called with null. This mean that we *can* actually add elements to a List⟨?⟩, but only nulls.

In general, a wildcard should not be considered the name of a type. For instance, the two occurrences of '?' in Pair⟨?,?⟩ are not assumed to stand for the same type, and even for the above list, 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⟩().

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 express that the list element type 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 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 whose element type is a subtype of Number. Again, 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: List⟨? **extends** Integer⟩ is a subtype of List⟨? **extends** Number⟩.

While extends-bounds introduce *upper* bounds on wildcards, *lower* bounds can be introduced using so-called super-bounds. So List⟨? **super** String⟩ is a supertype of List⟨*T*⟩ when *T* is a supertype of String, such as List⟨String⟩ and List⟨Object⟩.

This is useful, e.g., with Comparator objects. The Java platform class TreeSet represents a tree of ordered elements. 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 must provide a Comparator to compare Strings. This can be done by a Comparator⟨String⟩, but a Comparator⟨Object⟩ will do just as well, since Strings are Objects. The type Comparator⟨? **super** String⟩ is appropriate here, 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⟩.

Nested Wildcards

Generic classes can be nested, as in List⟨List⟨String⟩⟩, denoting lists of lists of strings. In this case List⟨String⟩ is a type parameter which is itself generic. Since generic instantiations with wildcards are just types, they too can be used as type parameters for other generic classes. Thus, the following code is legal:

```
List⟨List⟨?⟩⟩ lists = new ArrayList⟨List⟨?⟩⟩();
lists.add(new LinkedList⟨String⟩());
lists.add(new ArrayList⟨Integer⟩());
List⟨?⟩ list = lists.get(0);
```

List⟨List⟨?⟩⟩ is simply the type of lists of lists: an object of this type is a list that can contain all kinds of lists as elements. Hence, both lists of e.g. String and of Integer can be inserted, as shown. When extracting elements they have type List⟨?⟩, so we know the retrieved elements are lists, but we do not know their element type.

Type Inference with Wildcards

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:

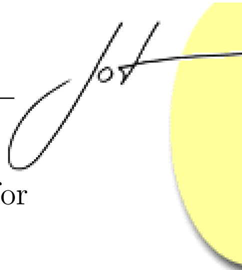
```
⟨T⟩ T choose(T a, T b) { ... }

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⟩. In GJ, different parameterizations of the same class are incomparable and the only such type is Object, even though Set⟨T⟩ and List⟨T⟩ share the superinterface Collection⟨T⟩. GJ is unable 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: 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—enough to, e.g., iterate and call toString() on its 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 infer Collection⟨? **extends** Number⟩.

3 WILDCARD CAPTURE

While wildcards provide a solution to a number of issues with parameterized classes, the simplistic mechanism described so far (which is largely derived from the use-site variance of Igarashi & Viroli [18]) does give rise to problems of its own. 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 one:

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

This method can be called with a Set⟨*T*⟩ for any type *T*, and it 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 follows:

```
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?

Our solution is linguistic: we 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. We therefore allow the call.

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.[1]

[1]The word "capture" is sometimes used to refer to the syntactic situation when free variables in an expression are brought into scope of a declaration of the same name. We do not believe that these two uses will clash, since they occur in largely separate domains.

Capturing wildcards is only legal in some situations. Intuitively, there must be a unique type to capture at runtime. So a type variable can only capture one wildcard, because the actual element types of, e.g., two different Set⟨?⟩s may be different. Also, only type variables that occur at "top level" in a generic class can be captured (as in Stack⟨T⟩ and unlike Stack⟨Stack⟨T⟩⟩ or Stack⟨T[]⟩). This is 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 effect of capture in this case is to allow the following call:

```
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. More generally, in the Java Platform API's we avoid providing a large number of duplicate methods having identical bodies, but different signatures.

Proper Abstraction

Wildcard capture also addresses a related problem. Consider the method Collections.shuffle(), which takes a list and shuffles its elements. One possible signature is as follows:

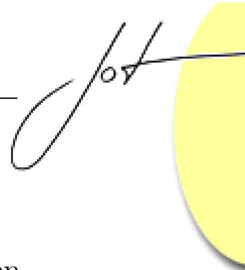
```
⟨T⟩ void shuffle(List⟨T⟩ list) { ... }
```

The type argument is needed because the method body needs a name for the element type of the list, to remove and re-insert 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.



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 [18]. Following this line of argument, the declaration of set as Set⟨?⟩ set on page 104 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 Y (a so-called *witness type*) for the particular element type of set *and* a name s for the set with the type Set⟨Y⟩. Both can then be used in a subexpression containing the method call to be captured. Using the syntax of Cardelli and Wegner [7] for existential expressions, we would have the interpretation:

```
∃X.Set⟨X⟩ set;
set = open set as s [Y] in
  pack [Z=Y in Set⟨Z⟩] Collections.⟨Y⟩unmodifiableSet(s);
```

Using this syntax it is clear that unmodifiableSet() is in fact called with a fixed type argument Y, because s has the type Set⟨Y⟩. Wildcard capture may therefore be seen as an implicit wrapping of polymorphic method calls with such **open** statements, when appropriate. However, there is more to it than that. In an **open** expression as above, it is disallowed for the witness type to "escape" its defining scope by occuring in the result type of the expression. Thus, in order to adequately express the capture semantics in terms of existentials, we must explicitly "repack" the result value of the expression to hide the witness type. This produces a value of the type ∃Z.Set⟨Z⟩, which can freely escape the scope of the **open** operation to be assigned to the variable set. Thus, the net effect is that the original existential of the expression set "bubbles up" through its enclosing expressions to appear on the type of the outermost one.

The connection between existential types and wildcards is here explained in an informal manner. Work is ongoing to express formally the the typing and semantics of wildcards in the context of a calculus (an extension of Featherweight GJ [16]) with an existentially based type system.

Preservation of Type Information

The existential interpretation of capture allows us to address one of the less appealing aspects of use-site variance, namely the need to sometimes throw away type information in the return type of method calls. Given a method whose return type includes a type parameter of the surrounding class, if for a given object that class is parameterized with a wildcard it is not obvious how to find the type of a call to that method. Indeed, given the approach of [18]), there are pathological cases where no unique most specific type exists, and an arbitrary choice must be made.

Even if a best choice exists, it may not be precise, and potentially causes the loss of important type information. Consider a class C and a variable c:

```
class C⟨X⟩ { Pair⟨X,X⟩ m() { ... } }
C⟨?⟩ c;
```

What is the type of the method call c.m() ? Under the approach of Igarashi and Viroli, the answer would be Pair⟨?,?⟩, loosing knowledge of the fact that the two element types of the pair are identical. However, with our "open-and-repack" existential interpretation above we can do better, expressing the type as ∃Y.Pair⟨Y,Y⟩, which retains all the information that we have. This precision can be exploited in a surrounding method call by using capture. For instance, the following method n() can be called as n(c.m()), which in turn would have the result type ∃Y.List⟨Y⟩:

```
⟨Z⟩ List⟨Z⟩ n(Pair⟨Z,Z⟩) { ... }
```

Thus, the "bubbling up" removes the problems of the type approximation in use-site variance and the imprecision that it introduces.

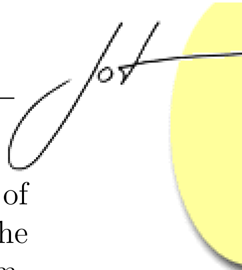
It does however mean that expressions can have types (such as ∃Y.Pair⟨Y,Y⟩) which cannot be expressed in surface syntax. From a puristic point of view this should be avoided in the design of programming languages.

We have chosen a more pragmatic approach. The full syntax of existential types is simply to involved to introduce in a mainstream programming language such as Java. Wildcards are a much more lightweight construct which can be understood in its own right, also by the vast majority of programmers not familiar with the theory of existential types. One should note the syntactic and conceptual difference between the actual Java syntax of a simple method call and the heavyweight explicit open-repack outlined above. On the other hand, we see no reason to artifically restrict the class of programs accepted by the type checker by omitting capture, just because of a principle forbidding its internal type representation to be richer that the surface layer. This goes especially when there are so many good examples of the usefulness of capture.

Also, the internal type system of Java is already enriched for other purposes, for instance with a form of intersection types. So in that sense the battle is already lost. Finally, as the next section shows, even with capture the compiler does not in fact need to fully represent existential types.

4 THE IMPLEMENTATION OF CAPTURE

Apparently, in order to implement capture a compiler for the Java programming language would have to be extended to handle existential types in general. Fortunately, this is not necessary: it turns out that there is a very simple strategy for implementing the above semantics—the process described in the Java Language



Specification [14] as *capture conversion*. In essence, all wildcard type arguments of the types of expressions are replaced by a fresh, synthetic type variable with the appropriate bounds. Type assignment and type inference (in the JLS and the compiler) is already equipped to deal with (explicitly declared) type variables in the type of expressions, and to propagate them as necessary to surrounding expressions, and it essentially requires *no* extra effort to deal with the synthetic variables introduced by capture conversion. Thus, while a full theoretical account based on existential types will need to deal with the explicit "bubbling up" of type variables in a rather verbose syntactic manner, in the implementation of a compiler this is actually much simpler to deal with than the type approximation approach proposed in [18].

Implementation of capture conversion

As mentioned, the implementation of wildcard capture introduces a fresh, bounded type variable to represent each actual type argument involving a wildcard, for each expression evaluation. It is required that we use distinct, fresh type variables even with two evaluations of the *same* expression as with x in x.foo(x), because the meaning of the fresh type variable is associated with a particular object, not a particular expression, and the same expression may evaluate to different objects on different occasions.

A method, Type capture(Type), is applied in order to transform top level wildcard type arguments into fresh type variables. In pseudo-code it works as follows:

```
Type capture(Type type) {
  if (type instanceof ClassType) {
    List⟨Type⟩ newArgs = new List⟨Type⟩();
    for (Type arg : type.typeParameters()) {
      if (arg instanceof WildcardType)
        newArgs.add(new TypeVariable());
      else newArgs.add(arg);
    }
    Iterator⟨Type⟩ iter = newArgs.iterator();
    for (Type arg : type.typeParameters()) {
      Type newArg = iter.next();
      if (arg instanceof WildcardType)
        convert(arg, newArg, newArgs);
    }
    return new ClassType(type.className, newArgs);
  } else {
    // a non-classType has no wildcard type arguments
    return type;
  }
}
```

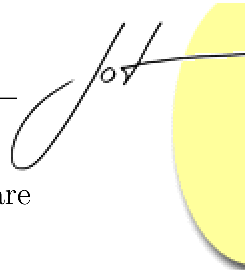
The effect is that capture(T) is the same as the type T except that convert has been applied on every top-level wildcard type argument. The actual implementation of capture is somewhat more complex, but the above pseudo-code presents the core which is relevant in connection with the type capture operation. Next, convert works as follows (also slightly simplified compared to the actual implementation):

```
void convert(WildcardType wc, TypeVariable newArg, List⟨Type⟩ newArgs)
{
  // bound of the formal type parameter corresponding to this wildcard
  Type formalBound = wc.getFormal().getBound();
  // substitute actuals for formals in formalBound, use as upper bound
  newArg.upper = subst(formalBound, wc.getFormals(), newArgs);
  if (wc matches ? extends T)
    newArg.upper = greatestLowerBound(T,newArg.upper);
  newArg.lower = wc.getLowerBound();
}
```

The method convert receives a WildcardType and uses it (including the knowledge about its corresponding formal type parameter) to equip the given TypeVariable with a lower bound and an upper bound. The semantics of this is that such a TypeVariable X stands for some (unknown but fixed) type which is a subtype of the upper bound and a supertype of the lower bound. The wildcard is used as one of the arguments in a type application (an expression where a type parameterized class or interface S receives actual type arguments), and by looking up the declaration of S we can find the formal type parameter and its bound. This has been done already, and the bound of the formal is available as wc.getFormal().getBound() above. Moreover, the wildcard may have its own bound (such as T if the wildcard is on the form '? extends T'). Consequently, the method convert finds the tightest possible upper and lower bounds for the new type variable by using the declared upper bound on the formal type argument and the upper bound on the wildcard itself (if present), respectively the lower bound on the wildcard (if present). Note that the formal type argument cannot have a declared lower bound.

This capture conversion process—transforming top-level wildcard type arguments into fresh type variables with suitable bounds—is applied whenever the type of an expression is computed, specifically with the following 7 kinds of expressions: conditional expressions (b? x:y), method invocations, assignment expressions, type cast expressions, indexed expressions (myArray[index]), select expressions (myObject.field), and identifier expressions (myName). Finally, capture conversion is also applied to the left-hand side of subtype relations:

```
boolean isSubType(Type x, Type y) {
  x = capture(x);
  //...
}
```



This is required in order to ensure that wildcards in covariant result types are captured correctly in connection with method overriding.

5 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, outline the origins of existential types and some connections to our work.

Virtual types

Wildcards ultimately trace their origins back to the language BETA [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, 17], 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,

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

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

$$\text{List}\langle\text{Integer}\rangle <: \text{List}\langle ?\ \textbf{extends}\ \text{Number}\rangle <: \text{List}\langle ?\rangle$$

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 in essentially the same manner as an add() call to a List⟨? **extends** Number⟩.[2]

Thus, virtual types in BETA is the first mechanism that lets different parameterizations of a generic class share an instance of that generic class as a common supertype in a statically safe manner. 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⟨?⟩.

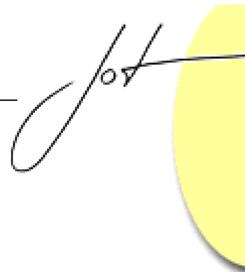
The gbeta language [13], which generalizes BETA in several ways, reduces the latter problem by having structural subtyping at the level of mixins, but the inheritance hierarchy must still be carefully planned and centrally managed. However, inspired by various variance mechanisms including [30] the notion of *constrained virtuals* has recently been added to gbeta, thus providing a 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] and the ν_{obj} calculus underlying the Scala language [26].

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:

[2] 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].



```
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.

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 [18], Igarashi and Viroli propose a significant extension, 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 [16], 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. A formalization that covers all the features is ongoing work.

Unlike wildcards, use-site variance relies 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.

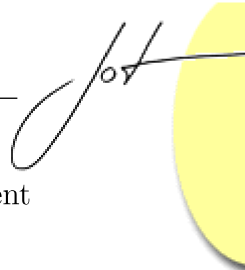
Wildcards focus instead on the type information trade-off: The less you require in a type, the more objects can be typed by that type. For example, the type List⟨? **super** Integer⟩ describes a larger set of objects than the type List⟨Integer⟩. In both cases it is harmless to call a read method like 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.

Existential Types

We have mentioned the connection between wildcards and existential types several times. Essentially, we consider wildcards as a language construct in its own right which provides a subset of the expressive power of full existential types, but with a more concise syntax and with some restrictions on the complexity of type expressions. In particular, it is not possible to choose freely where to put the existential quantifiers in the existential type that corresponds to a type expression using wildcards—for each '?' the corresponding quantifier will be on the immediately enclosing parameterized class. This ensures compositionality (a List⟨List⟨?⟩⟩ will actually contain elements of type List⟨?⟩), but of course it is impossible to express some types directly.

Existential types were introduced by Mitchell and Plotkin in [23], motivated by the desire to provide types for the values defined by Ada generic packages, CLU clusters and other abstract data type declarations, thus making these values first-class and allowing them to be passed as parameters etc. Infinite sums in category theory are used to illustrate the semantics of expressions having existential types, with the intuition that they represent an infinitary version of finite sums such as variant types.

Soon after, Cardelli and Wegner describe existential types with bounds in [7], expanding the focus on the interplay between parametric and subtype polymorphism. ML modules, in particular when constraining a structure with a signature, are mentioned as an example of using existential and dependent types, a line of work which has later been expanded significantly, e.g., by Russo in [27] where modules are first-class entities. The language Scala has been extended with dependent types, based on abstract type members as described and formalized in [26], and with some



inspiration from the BETA related languages. Wildcards do not support dependent types, even though we often discussed how to extend them to do it.

The connection between genericity in the Java language and existential types have emerged before. In [15], the so-called raw types of GJ are described as being close to existential types, but then formalized by giving the bottom type for unknown type arguments. That paper also mentions that Pizza uses existential types internally, in special casts which are similar to raw types in GJ. As mentioned before, Igarashi and Viroli [18] refer to existential types to establish an intuition about use-site variance. The difference to wildcards is that by capture they allow for using a method body in a similar way as the body of an **open** expression, whereas the calculus in [18] performs the equivalent of an **open** and a **pack** operation in one step, prohibiting usages of the witness type.

6 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. In conclusion, the expressive power of wildcards is a non-trivial enhancement to the language, essentially because wildcards provide much of the power of existential types without the complexity.

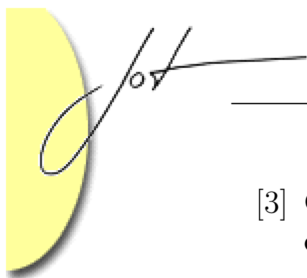
ACKNOWLEDGMENTS

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, Bob Deen, and Graham Hamilton who all had a significant influence on the resulting design.

REFERENCES

[1] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the java programming language. In *Proceedings OOPSLA* 1997. Toby Bloom, editor.

[2] P. America and F. van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings OOPSLA/ECOOP* 1990, pages 161–168. Norman K. Meyrowitz, editor.



[3] G. Bracha and D. Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *Proceedings OOPSLA* 1993. 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 *Proceedings OOPSLA* 1998. Craig Chambers, editor.

[5] K. Bruce. Subtyping is not a good match for object-oriented programming languages. In *Proceedings ECOOP* 1997. Mehmet Akşit and Satoshi Matsuoka, editors.

[6] K. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *Proceedings ECOOP* 1998. Eric Jul, editor.

[7] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. In *ACM Computing Surveys*, 17.4, pages 480–521. 1985.

[8] R. Cartwright and G. L. Steele. Compatible genericity with runtime-types for the Java programming language. In *Proceedings OOPSLA* 1998. Craig Chambers, editor.

[9] W. Cook. A proposal for making Eiffel type-safe. In *Proceedings ECOOP* 1989. Stephen Cook, editor.

[10] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings POPL* 1990. Paul Hudak, editor.

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

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

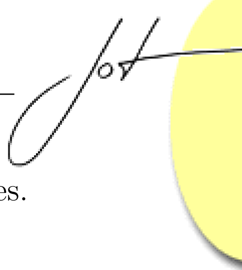
[13] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, 1999.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2004. To appear.

[15] A. Igarashi, B. Pierce, and P. Wadler. A Recipe for Raw Types. In *Informal Proceedings FOOL*, 2001.

[16] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Proceedings OOPSLA* 1999, pages 132–14. Linda Northrop, editor.

[17] A. Igarashi and B. C. Pierce. Foundations for virtual types. In *Proceedings ECOOP* 1999. Rachid Guerraoui, editor.



[18] A. Igarashi and M. Viroli. On variance-based subtyping for parametric types. In *Proceedings ECOOP* 2002, pages 441–469. Boris Magnusson, editor.

[19] A. Kennedy and D. Syme. Design and implementation of generics for the .NET common language runtime. In *Proceedings PLDI* 2001, pages 1–12. C. Norris and J. J. B. Fenwick, editors.

[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 *Proceedings OOPSLA* 1986, pages 391–405. 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 type. *Proceedings POPL* 1985, pages 37–51. Revised version in *ACM Transactions on Programming Languages and Systems*, 10.3, 1988, pages 470–502.

[24] A. Myers, J. Bank, and B. Liskov. Parameterized types for Java. In *Proceedings POPL* 1997. Neil D. Jones, editor.

[25] M. Odersky and P. Wadler. Pizza into Java: Translating theory into practice. In *Proceedings POPL* 1997, pages 146–159. Neil D. Jones, editor.

[26] M. Odersky, V. Cremet, C. Röckl, and M. Zenger, A Nominal Theory of Objects with Dependent Types. In *Proceedings ECOOP* 2003. Springer-Verlag.

[27] C. V. Russo. First-class structures for standard ML. *LNCS 1782*, pages 336++, Springer Verlag 2000.

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

[29] K. K. Thorup. Genericity in Java with virtual types. In *Proceedings ECOOP* 1997, pages 444–471. Mehmet Akşit and Satoshi Matsuoka, editors.

[30] K. K. Thorup and M. Torgersen. Unifying genericity. In *Proceedings ECOOP* 1999, pages 186–204. Rachid Guerraoui, editor.

[31] M. Torgersen. Virtual types are statically safe. In *Informal Proceedings FOOL*, 1998. K. Bruce, editor,

[32] M. Torgersen. *Unifying Abstractions.* PhD thesis, Computer Science Department, University of Aarhus, 2001.

ABOUT THE AUTHORS

Peter von der Ahé, M.Sc. was until recently at the University of Aarhus, Denmark. He now maintains javac at Sun Microsystems. Peter's home page is at http://www.ahe.dk/peter/.

Gilad Bracha, Ph.D. is a Computational Theologist at Sun Microsystems. He is co-author and maintainer of the Java language specification. His home page is at http://bracha.org/.

Erik Ernst, Ph.D. is a research associate professor at the University of Aarhus, Denmark. His home page is at http://www.daimi.au.dk/~eernst/.

Neal Gafter, Ph.D. works at Google Inc. He maintained javac at Sun Microsystems, Inc., until recently. His home page is at http://www.gafter.com/~neal/.

Christian Plesner Hansen, M.Sc. just graduated from the University of Aarhus, Denmark, and is now employed at OOVM a/s, Aarhus, Denmark. His home page is at http://www.daimi.au.dk/~plesner/.

Mads Torgersen, Ph.D. works as a research assistant professor at the University of Aarhus, Denmark. His home page is at http://www.daimi.au.dk/~madst/.

Moreover, the authors constitute the team which specified and implemented wildcards in the Java programming language.