Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

On the reification of Java wildcards[☆]

Maurizio Cimadamore, Mirko Viroli [*]

Alma Mater Studiorum, Università di Bologna DEIS, via Venezia 52, 47023 Cesena, Italy

ARTICLE INFO

Article history:
Available online 16 July 2008

Keywords:
Generic types
Wildcards
Java
Reification

ABSTRACT

Providing runtime information about generic types – that is, reifying generics – is a challenging problem studied in several research papers in the last years. This problem is not tackled in current version of the Java programming language (Java 6), which consequently suffers from serious safety and coherence problems. The quest for finding effective and efficient solutions to this problem is still open, and is further made more complicated by the new mechanism of wildcards introduced in Java J2SE 5.0: its reification aspects are currently unexplored and pose serious semantics and implementation issues.

In this paper, we discuss an implementation support for wildcard types in Java. We first analyse the problem from an abstract viewpoint, discussing the issues that have to be faced in order to extend an existing reification technique so as to support wildcards, namely, subtyping, capture conversion and wildcards capture in method calls. Secondly, we present an implementation in the context of the EGO compiler. EGO is an approach for efficiently supporting runtime generics at compile-time: synthetic code is automatically added to the source code by the extended compiler, so as to create generic runtime type information on a by need basis, store it into object instances, and retrieve it when necessary in type-dependent operations. The solution discussed in this paper makes the EGO compiler the first reification approach entirely dealing with the present version of the Java programming language.

1. Introduction

The long awaited extension of Java with *generics* has been shipped since J2SE 5.0 after several years of research and development, and currently represents the most substantial Java extension so far. However, generics are implemented without a runtime support, namely, there is no *reification* of them during execution. This means that generic types are simply introduced as a compile-time abstraction to enforce type-safety; they are *erased* to standard types by the compilation process, and hence they never enter the runtime domain of the Java Virtual Machine (JVM). As described in detail in [2,22] this makes generics (i) hardly fit important Java frameworks such as Serialization and Reflection; and (ii) differ from standard Java types as far as type-dependent operations are concerned (cast conversions, type tests through `instanceof` operator, array operations). But most importantly, erasure causes the so-called *heap pollution* problem: certain cast operations are statically accepted (with a warning) and succeed at runtime, but can later cause any field access or method invocation to unpredictably fail. Runtime generics are also shown to be intrinsically more expressive in supporting the development of certain programming idioms—like the *expression problem* [28]. Also, runtime generics are already implemented in competitor frameworks such as .NET [15,26,36].

[☆] This is an extended version of a paper that appeared earlier in Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07), Seoul, Korea, March 11-15, 2007; Marjan Mernik acted as guest editor for this paper.

[*] Corresponding author.
E-mail addresses: maurizio.cimadamore@unibo.it (M. Cimadamore), mirko.viroli@unibo.it (M. Viroli).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.06.005

Several solutions have been studied to address this problem—a rather complete list of references is [19,1,4,22,14,13,34,32,33,20,2]. On the one hand, the JVM can be redesigned to directly represent generic types [14], possibly limiting the modifications to the class loader only, as shown e.g. in [1]. On the other hand, generics can be reified by the compiler, which can automatically create additional code running on the existing JVM, and simulating a layer of runtime generics. Two approaches following this direction are NextGen [4,20] and EGO (Exact-Generics on Demand) [33]. Though a JVM-based approach inevitably leads to a better-engineered solution (with greatest performance and coherence), as developed e.g for .NET [26], compile-time approaches are still an interesting alternative that require no JVM modification and is more easily deployed.

Most of the work on reification developed so far is based on generics as introduced in GJ language [19], that is, they consider class-based, F-bounded polymorphism [6]. However, the Java programming language now comes equipped with a brand new mechanism related to generics, called *wildcards* [30]. This is the result of applying the construct known as *variant parametric types* (VPTs) to the Java programming language [9,35]. Wildcard types (WTs) (also called wildcard parameterised types) are types of the kind `List<? extends T>`, `List<? super T>`, `List<?>`—where `T` can be any valid reference type. WTs can be considered as a notation to abstract over a number of different instantiations of the same generic class, e.g. any `List<T>` where `T` is subtype of `Number` can be passed to where a `List<? extends Number>` is expected. This construct finds many suitable applications, e.g., in the Java Collections Framework (JCF) and the the Java Reflection API; in general, wildcards provide a means by which subtyping (inclusive polymorphism) can better integrate with generics (parametric polymorphism).

Reifying WTs is currently an unexplored problem. It happens that they lead to very subtle semantic issues [9,35,30], for they feature a multi-variant subtyping structure, (partially) hide a type system based on existential types, and affect type inference in generic method calls. Hence, existing reification proposals can deal with wildcards only after a substantial redesign that tackles very subtle and tricky issues. We strongly believe that this problem is of key importance: its solution would lead to a new version of the Java programming language where generics and wildcards are handled as first-class types, and are more homogeneously and uniformly integrated in the language.

The goal of this paper is to discuss a runtime implementation support for wildcard types in Java. The first contribution is to analyse all the issues arising when reifying wildcards, and identify viable solutions for addressing them which could be applied to any existing reification technique. This is achieved by considering the research theory on wildcards, as well as details of the Java Language Specification (JLS) [10] and the current Java compiler, and identifying all the semantic issues that have to be taken into account to provide a reification support—discussing e.g. non-termination of subtyping, hidden existential types, and so on. Secondly, we apply this analysis to a specific reification technique, namely the EGO compiler [33]. In EGO, the compiler automatically creates the necessary code to make sure that generic type information is created once and on a by need basis, it is stored into object instances, and – thanks to specific caching techniques – it is quickly retrieved when necessary in type-dependent operations. We show how the EGO project has been smoothly extended to deal with wildcards, leading to the first reification approach for Java generics that entirely deals with the Java programming language (from version 5.0).

The remainder of the paper is organised as follows. Section 2 summarises Java wildcards and analyses the reification problem, Section 3 provides a detailed discussion of possible solutions, Section 4 briefly describes the EGO compiler, Section 5 explains how this has been extended to deal with WTs, and finally Section 6 concludes providing final remarks.

2. On wildcards and reification

2.1. The wildcards mechanism in Java

Generics are useful to parameterise a class in a type: by providing different instantiations for such a type, several classes are obtained that are useful in different contexts. For instance, a generic collection class `Collection<E>` can be defined that abstracts over the type `E` of its elements. Type variable `E` is later to be instantiated to an actual type, so that types such as `Collection<String>` and `Collection<Integer>` can be used where necessary.

There are some situations however in which only partial knowledge about the instantiation of a type variable is required, hence no instantiation of it is a good choice. Suppose that a method `containsAll(Collection<E> c)` is to be added to class `Collection<E>`, which takes another collection `c` and checks whether its elements are all contained in the receiver. When this method is invoked on a receiver with type `Collection<Number>`, only another collection of type `Collection<Number>` can be passed as argument, though it is easy to recognise that also instances of `Collection<Integer>` and `Collection<Float>` could in principle be passed—supposing `Integer` and `Float` are subtypes of `Number`. In general, any type `Collection<T>` where `T` is a subtype of `Number` could be passed, but this is not possible using standard generics for they are invariant—`Collection<Integer>` is not a subtype of `Collection<Number>` [9].

This situation can be overcome by integrating parametric polymorphism (generics) and inclusive polymorphism (subtyping) [9], as developed in the wildcards mechanism introduced in Java 5.0. After class `Collection<E>` has been defined, one can use a type of the kind `Collection<? extends E>`, called a *bounded wildcard (parameterised) type*, as follows:

```
class Collection<E> {
 ...
 boolean containsAll(Collection<? extends E> c){..}
}
```

```
...
Collection<Number> cn=new Collection<Number>(...);
Collection<Integer> ci=...
Collection<Float> cf=...
boolean b1=cn.containsAll(ci);
boolean b2=cn.containsAll(cf);
```

Type `Collection<? extends E>` factors over any `Collection<T>` where `T` is a subtype of `E` (written `T<:E` henceforth): hence, any object of such a type can be passed, resulting in an enhanced applicability of the method. The following example, again taken from the JCF, shows another kind of wildcard:

```
class Collections {
 ...
 <T> static void sort(List<T> l,Comparator<? super T> c){..}
}
...
List<Integer> li=...;
Comparator<Integer> ci=...;
Comparator<Number> cn=...;
sort(li,ci);
sort(li,cn);
```

Here, the static `sort()` method accepts as input a list of elements with type `T` to be sorted, and a comparator for such elements. Instead of declaring the comparator's type as being `Comparator<T>`, it is more useful to use `Comparator<? super T>`: any instance of a type `Comparator<S>` with `T<:S` can be passed – dually to the case "`? extends T`" –, e.g. a comparator of `Number`s can be used to compare `Integer`s. The last example of wildcard type is the unbounded version `List<?>`, literally meaning *any* `List<T>`, which is used when the actual type of the list element does not care, as in a method of the kind:

```
static int size(Collection<?> c){..}
```

All such new types find an extensive use in the JCF to define in a flexible way constraints on the parameterisation of collections. These types cannot be used to create objects in `new` expressions (expression `new List<?>(..)` is disallowed); rather, they can be thought of as sort of interfaces over standard generic types. WTs can in fact be understood as a generalisation of standard generic types, where the type parameter is not a concrete type, but rather a set of types, similar to a sort of "interval"—though it is not a proper interval for the subtype relation is a partial order. In particular, a type `List<? extends T>` is associated to interval [`<nulltype>`, `T`], for it is a supertype of any `List<R>` where `R` is in between `<nulltype>` and `T`—namely, `R` is a supertype of `<nulltype>` and a subtype of `T`. Similarly, type `List<? super T>` is associated to interval [`T`, `Object`], `List<?>` to [`<nulltype>`, `Object`], and finally non-wildcard type `List<T>` to [`T`, `T`]—supposing `Object` is the bound of `X` in the definition of class `List<X>`.

Most semantic aspects of WTs can be described in terms of this interval-like notion, which in fact resembles the interpretation of VPTs in terms of existential types [9]. For instance, subtyping is intuitively expressed in terms of inclusion of such intervals: given two WTs, `S` and `T`, `S` is a subtype of `T` if the interval induced by `S` is included in the one of `T`. For instance we have:

```
List<? extends Integer> l1 = ...;
List<? extends Number> l2 = l1; // Covariance of "? extends"
// OK, for [<nulltype>,Number] includes [<nulltype>,Integer]

List<? super Number> l3 = ...;
List<? super Integer> l4 = l3; // Contravariance of "? super"
// OK, for [Integer,Object] includes [Number,Object]

List<? super Integer> l5 = ...;
List<? extends Number> l6 = l5; // Compile-time error
// No, for [<nulltype>,Number] does not include [Integer,Object]
```

2.2. On the reification problem

In this section we analyse the reification problem from which Java currently suffers, and point out abstract requirements of a complete solution to the problem.

The syntax of reference types in Java is informally expressed (following typical conventions of [6]) as follows:

```
W ::= T | ? extends T | ? super T | ?          // Argument type
T ::= I<W1,..,Wn> | C<W1,..,Wn> | T[]          // Reference types
R ::= C<> | C<?,..,?> | I<> | I<?,..,?> | R[] // Reifiable types
K ::= C<T1,..,Tn> | R[]                        // Types of objects
```

where `C` ranges over class names, `I` over interface names, and lists such as "`W1`,…,`Wn`" can be void ($n \geq 0$). Reference types `T` can thus include a (possibly generic) interface `I<W1,..,Wn>`, a class `C<W1,..,Wn>`, and an array `T[]`.[1]

We first note that not all such types can be used to create objects, rather, Java operator `new` can be used only for types `K`. On one hand, as already mentioned, a generic class can be used provided it has no wildcards (at the top-level), otherwise it would be a sort of interface—a type like `List<List<?>>` could be used however. On the other hand, some restrictions are applied to arrays: they can be created only if their element type `R` is a so-called `reifiable` type [10], namely, it is an array whose element type is a reifiable type—either a class (or interface) type with zero type arguments or with the unbounded wildcards everywhere.

Reifable types are also the only types which can be used in type tests with the `instanceof` operator, namely, the only types which the runtime system can "see" when inspecting an object. This is a crucial point: in Java there is a mismatch between those types `K` that are available at compile-time to create objects and types `R` that are available at runtime for inspection. After the generic type of an object is lost at compile-time for some reason, due e.g. to an upcast to `Object`, there is no way of recovering the instantiation of type parameters, for they are not available for inspection:

```
Object o=new List<String>(..);
boolean b=o instanceof List<?>; // true!
boolean b=o instanceof List<String>; // Compile-time error
```

So, in Java not all types are *reified*, for the erasure process during Java compilation [19,10] drops any generic information from the bytecode, and hence from the runtime.

This known problem causes also the so-called *heap pollution* of Java [10]. The generic type of an object cannot be safely recovered, hence it might be possible that – by unchecked conversions – the compile-time representation of an object does not actually fit the content of the heap, later resulting in an exception in unexpected places of the code, as in the following example:

```
List<Integer> li=new List<Integer>(..);
Object o=li; // To be used e.g. in Serialization
...
List<String> ls=(List<String>)o; // Unchecked warning
...
String s=ls.get(0); // Raises a ClassCastException
```

Heap pollution is a significant shortcoming for the Java programming language. Even the Java API has been compiled with unchecked warnings: this means that any program exploiting the API can in principle incur in exceptions in unexpected places of the code. It is virtually impossible to certify a Java 5.0/6 code to be free of heap pollution: the language safety that generics aimed at promoting is therefore compromised.

The goal of an extension of the language fully-reifying generic information is hence to solve this problem, making the distinction between reifiable and non-reifiable types vanish. The new syntax of types would be the following:

```
W ::= T | ? extends T | ? super T | ?  // Argument type
T ::= I<W1,..,Wn> | C<W1,..,Wn> | T[]  // Reference types
K ::= C<T1,..,Tn> | T[]                // Type of objects
```

Hence the goal of providing a full reification support to generics and wildcards is to lead to a more coherent and safe version of the Java programming language.

2.3. On Java wildcards specification

In order to provide a full reification support tackling the Java wildcards mechanism, it is of great importance to have a precise, clean, and complete description of their semantics available—in principle, a whole formalisation would be needed. However, at this time this is not the case.

A first reason comes from the origins of this mechanism. After an initial attempt to introduce VPTs [8] in Java failed,[2] a similar expressiveness and rather different syntax was obtained through a fairly new mechanism of wildcards (more inspired to structural virtual types [27]). A main difference was on the way wildcard types are used when accessing fields/methods/supertype through them (i.e. opening/closing [9]): although the wildcards solution is more expressive, it ultimately exposes Java programmers with the difficult concept of *existential type* [16,35,29,3] —hidden by the *capture conversion* mechanism as shown in next section (see also [10] Section 5.1.10). This new semantics leads to a language mechanism that has no clear and complete formal foundation. One work trying to formalise this mechanism is the workshop paper in

[1] Actually, another kind of types are *raw types* [7], introduced in Java 5.0 to let programmers use generic types in the legacy way, as unparameterised types. In this paper we decided to neglect their treatment for the sake of simplicity. How/whether they should be supported in a language with reification is a completely orthogonal issue.

[2] It temporarily appeared in the so-called *Tiger* prototype of J2SE5.0 compiler (see [21]).

[29], which defines a core calculus for wildcards along with a type system based on existential types. However, in this work crucial parts of the semantics are neglected, like type inference in method calls, and no proof of properties is reported. Similarly, in [3] a subset of wildcards that does not deal with lower bounds is formalised using existential types and is proven sound. These works might be a solid basis for fully formalising wildcards, but this objective is currently not achieved.

Similarly, the JLS mostly underspecifies the semantics of wildcards, which is moreover even subject to changes as new (sub)versions are released, making it difficult to systematically analyse, describe, and use them. Very often, testing the behaviour of JDK compiler – or even trying to interpret its source code – is the only means to check how certain subtle situations are dealt with.

Accordingly, starting from all the above sources of information, in next section we analyse what subtleties of wildcards may affect the reification problem, and propose solutions which can result in an easier understanding of the mechanism.

3. Issues in reifying wildcards

Starting from the problem setting described in previous section, we discuss in detail all the issues that must be tackled in order to extend an existing reification support for generics so as to deal with the wildcards mechanism.

3.1. Representation of wildcards

A first problem to be faced is obviously how to represent WTs at runtime. An existing reification support for generics should already provide means to keep a representation of types that does not mention wildcards, like e.g. `C<String>`, `D<String,Integer>`, and so on. In order to deal with wildcards, it should be possible to represent also types of the kind `C<C<?>>`, `D<C<? extends Numer>,C<? super Number>>`, and the like, that is, possibly having wildcards at some inner level of parameterisation. For uniformity with the recursive case, the runtime system should deal with the general situation of generic types with the four kinds of parameterisation `C<T>`, `C<? super T>`, `C<? extends T>`, and `C<?>`; moreover, for any type `T` it should be able to represent the array `T[]`. Concerning arrays, and as far as runtime support is concerned, we note that a type `T[]` could be simply represented as a class type `Array<? extends T>`, where `Array` is a special generic class with one type argument: this basically accommodates all the subsequent issues concerning integration of wildcards and arrays, such as e.g. subtyping between arrays.[3]

Concerning class types, there are two main ways in which they can be represented, which are both viable techniques for an actual reification proposal. On one hand, the conceptual framework of variant parametric types (VPT) [9] can be adopted. There, each type parameter of a generic type is associated with a flag `v`, or *variance annotation*, namely: `o`, `+`, `-` and `*`, used for denoting an invariant, covariant, contravariant and bivariant type parameter, respectively. A mapping between VPTs and WTs is defined so that `C<oT>` means `C<T>`, `C<+T>` means `C<? extends T>`, `C<-T>` means `C<? super T>`, and finally `C<*T>` means `C<?>` independently of `T`. For example, type `E<? extends String,? super Number,?>` can be represented by VPT-like notation `E<+String,-Number,*Object>`. Note that this representation subsumes the one for standard generic types, since a generic type can be represented as a VPT where all the annotation symbols are set to `o`, e.g. `E<Number,Number,String>` is represented by `E<oNumber,oNumber,oString>`.

Alternatively, the concept of *wildcard type argument* can be introduced as described in [10], leading to a slightly different representation of WTs. A wildcard type argument can be thought of as an unnamed type variable associated to both an upper and a lower bound; in this case, a mapping can be defined so that a WT can be represented by a generic type exploiting one or more wildcard type arguments. As an example, the type `List<? extends String>` can be represented as the generic type `List<W>` where `W` is a wildcard type argument whose upper bound is `String` and whose lower bound is `<nulltype>`. The other types of wildcard are handled similarly, assigning lower bound and upper bound considering the interval metaphor. This kind of representation is the one that naturally follows from the JLS, though in the following we will abstract away from which technique is actually used, for they are mostly equivalent ones.

Other than representing wildcards, the representation of a WTs must also keep track of the upper bound(s) that a class declaration associates to each type parameter. Given a definition of the kind "`class C<`X_1 `extends` T_1`,..,`X_n `extends` T_n`>`", type T_i is associated to the i^{th} type parameter of any type generated from `C`. Note that a bound could actually be a so-called F-bound [19], namely, could mention type variables of the class as in definitions `C<X extends B<X>>` or `C<X extends B<Y>,Y extends B<X>>`. A main reason for keeping track of such bounds is that they are the actual upper bound of wildcard type arguments for types of the kind `C<? super T>`.

After wildcards get a proper representation, the reification support should guarantee that an object created from a type exploiting a wildcard, like `List<List<? extends Number>>`, is associated to the runtime representation of such a type in some way, so that the below code can be correctly executed:

```
Object o=new List<List<? extends Number>>(..);
...
boolean b=o instanceof List<? extends List<?>>;
```

[3] The use of the bounded wildcard in `Array<? extends T>` is motivated by the fact that Java arrays are covariant, namely `T[]` is a subtype of `R[]` if and only if `T` is a subtype of `R`.

In order to process the `instanceof` operator, the runtime system must first access the type of object `o`, and then use it to check whether `List<List<? extends Number>>` is a subtype of `List<? extends List<?>>`. The way such an association is to be implemented in not an issue here, for it should likely mimic the way in which the existing reification support already associates objects to their types.

As for next subsections, we conclude each discussion by reporting the concrete requirements of a reification support for wildcards.

Requirement 1. The reification support should:

- attach to each object a type representation `T` of the kind `C<W1,..,Wn>` where each `W` is of the kind `T`, `? extends T`, `? super T`, or `?`;
- make such types carry type (F-)bounds to each type parameter;
- provide a means to retrieve the type of an object when needed in type-dependent operations.

3.2. Capture conversion and direct supertype

Other than representation, the above example shows that another crucial issue is subtyping: the runtime system must be able to check whether two types (possibly being WTs) are in the subtype relation. A first problem is finding the direct supertype of a WT, which is of course a crucial brick of subtyping test. The JLS ([10] section 4.10.3) states that:

> "The direct supertypes of the type $\mathtt{C}{<}\,\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n\,{>}$, where at least one of the $\mathtt{R}_i$, $\mathtt{1} \leq \mathtt{i} \leq \mathtt{n}$, is a wildcard type argument, are the direct supertypes of $\mathtt{C}{<}\,\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_n\,{>}$, where $\mathtt{C}{<}\,\mathtt{X}_1, \mathtt{X}_2, \ldots, \mathtt{X}_n\,{>}$ is the result of applying *capture conversion* to $\mathtt{C}{<}\,\mathtt{R}_1, \mathtt{R}_2, \ldots, \mathtt{R}_n\,{>}$."

This means that each time the direct supertype of a WT is to be computed, a conversion operator called *capture conversion* is applied. Such an operator basically amounts at dropping wildcard type arguments in favour of *fresh type variables* with certain bounds. Consider a generic class definition of the kind $\mathtt{C}{<}\,\mathtt{X}_1\ \mathtt{extends}\ \mathtt{T}_1, \mathtt{X}_2\ \mathtt{extends}\ \mathtt{T}_2, \ldots, \mathtt{X}_n\ \mathtt{extends}\ \mathtt{T}_n\,{>}$, and let notation $[\mathtt{T}_1/\mathtt{X}_1, \ldots, \mathtt{T}_n/\mathtt{X}_n]\mathtt{R}$ represent type `R` after all occurrences of type variable $\mathtt{X}_i$ are substituted with type $\mathtt{T}_i$ (for each i). Capture conversion operator accepts a (possibly wildcard) generic type of the kind $\mathtt{C}{<}\,\mathtt{W}_1, \mathtt{W}_2, \ldots, \mathtt{W}_n\,{>}$ and returns a new type $\mathtt{C}{<}\,\mathtt{V}_1, \mathtt{V}_2, \ldots, \mathtt{V}_n\,{>}$, obtained by substituting each wildcard type argument $\mathtt{W}_i$ with a fresh type variable (say it is `Z`) with certain lower bounds and upper bounds (denoted as $\Delta^-(\mathtt{Z})$ and $\Delta^+(\mathtt{Z})$, respectively). These are computed using the interval metaphor as follows:

- if $\mathtt{W}_i$ is of the kind `?` (i.e. an unbounded wildcard) then $\mathtt{V}_i$ is a fresh type variable `Z` such that $\Delta^+(\mathtt{Z})$ is $[\mathtt{V}_1/\mathtt{X}_1, \ldots, \mathtt{V}_n/\mathtt{X}_n]\mathtt{T}_i$ and $\Delta^-(\mathtt{Z})$ is `<nulltype>`;
- if $\mathtt{W}_i$ is of the kind $\mathtt{?\ extends\ B}_i$ then $\mathtt{V}_i$ is a fresh type variable `Z` such that $\Delta^+(\mathtt{Z})$ is the greatest type that is both a subtype of $\mathtt{B}_i$ and $[\mathtt{V}_1/\mathtt{X}_1, \ldots, \mathtt{V}_n/\mathtt{X}_n]\mathtt{T}_i$, and $\Delta^-(\mathtt{Z})$ is `<nulltype>`;
- if $\mathtt{W}_i$ is of the kind $\mathtt{?\ super\ B}_i$ then $\mathtt{V}_i$ is a fresh type variable `Z` such that $\Delta^+(\mathtt{Z})$ is $[\mathtt{V}_1/\mathtt{X}_1, \ldots, \mathtt{V}_n/\mathtt{X}_n]\mathtt{T}_i$ and $\Delta^-(\mathtt{Z})$ is $\mathtt{B}_i$;
- if $\mathtt{W}_i$ is a non-WT argument then $\mathtt{V}_i$ is $\mathtt{W}_i$.

In other words, thanks to lower bounds and upper bounds, a fresh type variable is allowed to range over all the possible instantiations expressed by the original wildcard. For instance, we have that capture conversion of `Pair<? extends String, Integer>` yields `Pair<Z,Integer>` where `Z` is a fresh type variable such that $\Delta^+(\mathtt{Z})$ = `String` and $\Delta^-(\mathtt{Z})$ = `<nulltype>`, also written `<nulltype><:Z<:String`—note that the second argument `Integer` is not affected by capture conversion, for it is not a wildcard.

Consider now a more complex example of a class definition of the kind "`class D<X> extends C<C<? extends X>>`": what would be the direct supertype of type `D<? extends String>`?. The first step is to apply capture conversion to `D<? extends String>`, which yields `D<Z>` where `<nulltype><:Z<:String`. Now, we simply compute the supertype of this new type – the standard way is used since this type has no wildcard type argument – which yields `C<C<? extends Z>>`, where `<nulltype><:Z<:String`. Notice that this type cannot be directly expressed by a programmer in Java—it is not a proper WT, it can be understood as the bounded existential type $\exists$`Z<:String.`$\exists$`Y<:Z.C<C<Y>>` as described in [35,29,10]. However, the Java compiler already deals with this kind of types. As a result, a reification support dealing with wildcards must be able to represent even such types, and in particular, the concept of fresh type variable (with an upper and a lower bound).

Note that the interplay between capture conversion and F-bounded polymorphism can lead to recursive patterns as the upper/lower bound of a fresh type variable might rely upon the fresh type variable itself. This can happen when capturing a WT $\mathtt{C}{<}\,\mathtt{W}_1, \mathtt{W}_2, \ldots, \mathtt{W}_n\,{>}$ in which some $\mathtt{W}_i$ is of the kind "`? super T`" or "`?`". For example, given a class declaration of the kind:

```
class D<X extends Comparable<X>> extends C<C<? extends X>> ..
```

the direct supertype of `D<? super Number>` is computed as follows

(1) Apply capture conversion to the type `D<? super Number>`, which yields type `D<F>`, where `F` is a fresh type variable with bounds $\Delta^-(\mathtt{F}) = \mathtt{Number}$ and $\Delta^+(\mathtt{F}) = \mathtt{Comparable<F>}$;

(2) the direct supertype of `D<? super Number>` is given by the type substitution `[F/X]C<C<? extends X>>`, hence it is `C<C<? extends F>>` where $\Delta^-(\texttt{F}) = \texttt{Number}$ and $\Delta^+(\texttt{F}) = \texttt{Comparable<F>}$.

Requirement 2. The reification support should:

- represent the concept of fresh type variable as possible instantiation of a type parameter, carrying a type (F-)upperbound and lowerbound;
- use it to compute the direct supertype of a type through capture conversion.

3.3. The subtyping algorithm

We here present a simplified and ready-to-implement version of the subtyping algorithm for wildcard types, obtained by a combination of three main ingredients: capture conversion, standard inheritance, and subtyping between existential types as shown in [29,9,3]. Let `R` and `S` be two correctly formed WTs of the kind $\texttt{C}<\texttt{U}_1, \texttt{U}_2, \ldots, \texttt{U}_m>$ and $\texttt{D}<\texttt{V}_1, \texttt{V}_2, \ldots, \texttt{V}_n>$, respectively ($m, n \geq 0$); the algorithm decides whether `R` $<:$ `S` in the following steps:

(1) while $\texttt{C} \neq \texttt{D}$, compute the test `R'` $<:$ `S` where `R'` is the direct supertype of `R`;
(2) when $\texttt{C} = \texttt{D}$ (then $n = m$), first convert by capturing both `R` and `S`, and then for all i ($1 \leq i \leq n$):
   - if $\texttt{V}_\texttt{i}$ is not a fresh type variable, check that $\Delta^-(\texttt{U}_\texttt{i}) = \Delta^+(\texttt{U}_\texttt{i}) = \texttt{V}_\texttt{i}$;
   - if $\texttt{V}_\texttt{i}$ is a fresh type variable, check that $[\texttt{U}_\texttt{i}/\texttt{V}_\texttt{i}]\Delta^-(\texttt{V}_\texttt{i}) <: \Delta^-(\texttt{U}_\texttt{i})$ *and* that $\Delta^+(\texttt{U}_\texttt{i}) <: [\texttt{U}_\texttt{i}/\texttt{V}_\texttt{i}]\Delta^+(\texttt{V}_\texttt{i})$;
(3) if $\texttt{C} = \texttt{Object} \neq \texttt{D}$ fails.

As already discussed, fresh type variables are hence handled as "intervals" between the lower bound and upper bound, while non-type variables are singletons: the two cases in (2) basically amounts at checking interval containment. As an example consider a class declaration of the kind `List<X>`; then the type `List<? extends Number>` is a subtype of `List<?>` since:

- by capturing them we obtain `List<X>` where `<nulltype>`$<:$`X`$<:$`Number`, and `List<Y>` where `<nulltype>`$<:$`Y`$<:$`Object`;
- the argument of second type being a fresh type variable, we use the second rule of (2), which is successful since $\Delta^-(\texttt{Y}) = \texttt{<nulltype>} <: \Delta^-(\texttt{X}) = \texttt{<nulltype>}$ and $\Delta^+(\texttt{X}) = \texttt{Number} <: \Delta^+(\texttt{Y}) = \texttt{Object}$.

As an example of subtyping involving F-bounds, consider a class declaration of the kind `List<X extends Comparable<X>>`; then the type `List<? super Number>` is a subtype of `List<? super Integer>` since:

- by capturing both sides we obtain (accordingly to the F-bounded variant of the capture conversion algorithm described in the previous section) `List<X>` where `Number`$<:$`X`$<:$`Comparable<X>`, and `List<Y>` where `Integer`$<:$`Y`$<:$`Comparable<Y>`;
- the argument of second type being a fresh type variable, we use the second rule of (2), which is successful since $\Delta^-(\texttt{Y}) = \texttt{Integer} <: \Delta^-(\texttt{X}) = \texttt{Number}$ and $\Delta^+(\texttt{X}) = \texttt{Comparable<X>} <: [\texttt{X}/\texttt{Y}]\Delta^+(\texttt{Y}) = [\texttt{X}/\texttt{Y}]\texttt{Comparable<Y>} = \texttt{Comparable<X>}$.

Requirement 3. The reification support should:

- implement the subtyping algorithm between two types using the above approach;
- rely on it to support type-dependent operations.

3.4. On decidability

Unfortunately, decidability of subtyping with wildcards is still an open issue [12,11,29]. Accordingly, the above subtyping algorithm, which mimics the one used at compile-time by current implementation of J2SE 6 compiler, might possibly fail to terminate. This is of course undesirable at compile-time, and it is even worse at runtime. It is possible to show that the execution of certain Java subtyping tests never terminates under the assumption that generics and wildcards are reified at runtime and the above subtyping algorithm is used. Fig. 1 shows two examples involving non-termination when performing subtyping tests. In the top side, the runtime should perform the subtyping test `B` $<:$ `A<? super B>`, recursively leading to the same test after few algorithmic steps, as shown below:

$$\begin{array}{c}
\texttt{B <: A<? super B>} \\ \hline
\texttt{A<A<? super B>> <: A<? super B>} \\ \hline
\texttt{A<A<? super B>> <: A<X>} \quad \texttt{B <: X} \\ \hline
\texttt{B <: A<? super B>} \\
\vdots
\end{array}$$

This non-termination problem can actually be easily prevented by detecting loops when performing the subtyping test, e.g. by exploiting a *subtyping cache* keeping track of all pending subtyping tests. When performing the subtyping test `S` $<:$ `T` the subtyping cache is searched for a matching pair of the form (S, T); if such a pair exists the test is aborted and the answer `false` is provided. In the example shown above, the runtime should detect that a matching pair for the subtyping test `B` $<:$ `A<? super B>` already occurs in the subtyping cache, so that the test can terminate immediately with a negative reply. Note that this is the way in which current Java compiler deals with this kind of non-termination [24].

```
class A<X> { }
class B extends A<A<? super B>> {
   public void m() {
      Object o1 = new B();
      Object o2 = (A<? super B>)o1;
   }
}
```

```
class A<X> { }
class B<X> extends A<A<? super B<B<X>>>> {
   public void m() {
      Object o1 = new B<Object>();
      Object o2 = (A<? super B<Object>>)o1;
   }
}
```

Fig. 1. Two examples of non-termination of the subtyping algorithm.

However, there can be more offending situations as described in the second example in Fig. 1. The subtyping test there goes as follows:

$$\begin{array}{c}
\texttt{B<Object>} <: \texttt{A<? super B<Object>>} \\ \hline
\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<? super B<Object>>} \\ \hline
\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<X>} \quad \texttt{B<Object>} <: \texttt{X} \\ \hline
\texttt{B<Object>} <: \texttt{A<? super B<B<Object>>>} \\ \hline
\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<? super B<B<Object>>>} \\ \hline
\texttt{A<A<? super B<B<Object>>>>} <: \texttt{A<Y>}, \texttt{B<B<Object>>} <: \texttt{Y} \\ \hline
\texttt{B<B<Object>>} <: \texttt{A<? super B<B<Object>>>} \\
\vdots \\
\texttt{B<B<B<Object>>>} <: \texttt{A<? super B<B<B<Object>>>>} \\
\vdots
\end{array}$$

Caching is per se not sufficient to detect this kind of non-termination since the subtyping test `B<Object>` $<:$ `A<? super B<Object>>` recursively leads to the more complicated expression $\texttt{S}_\texttt{i} <: \texttt{T}_\texttt{i}$ where both types `B<Object>` and `A<? super B<Object>>` are nested in `S` and `T`, respectively (`i` is the nesting level). As such, we cannot take advantage of a subtyping cache since types are growing endlessly. Note that `javac` suffers from a very similar problem [25], causing the compiler to crash – with a `java.lang.StackOverflowError` error – when given the above code the following instruction is compiled:

```
A<? super B<Object>> o = new B<Object>();
```

The only known way to tackle this problem is to force the subtyping algorithm to abort and yield a negative reply after a certain amount of steps (which is quite reasonable in a runtime perspective). In [12] the problem of subtyping decidability is formally characterized with respect to the so called declaration-site variance setting, that is, where subtyping properties of generics are expressed at declaration time like in C# and Scala, rather than when using a type as with wildcards (use-site [9]). Under this assumption subtyping decidability can be viewed as the result of the interplay between (i) contravariance, (ii) *non-finitary inheritance* – `extends`/`implements` clauses possibly leading to non-finite sets of direct supertypes – and (iii) multiple instantiation inheritance—implementing several instantiations of the same generic interface (e.g. `Comparable<Integer>` and `Comparable<String>`). In particular, the Scala language is equipped with declaration-site variance in a way that subtyping decidability is preserved [17], since non-finitary inheritance is disallowed at compile-time by means of a technique derived from [31].

Unfortunately no complete solution has been proposed for Java so far. Java forbids multiple instantiation inheritance (because of type erasure), but this is not sufficient for proving subtyping decidability, since Java relies on use-site variance, which is more expressive and powerful with respect to declaration-site variance. Moreover, Java supports F-bounded polymorphism, possibly leading to forms of non-termination of subtyping that have not been characterised yet.

Requirement 4. The reification support should:

- ensure termination of the subtyping test (possibly yielding a negative reply when a proper solution is not likely found).

```
class ListUtils {

   public static List<?> clone(List<?> l) {
      return doClone(l);
   }

   private static <T> List<T> doClone(List<T> l) {
        List<T> newList=new List<T>();
        newList.head=l.head;
        newList.tail=l.tail;
        return newList;
   }
   ...
}

List<?> l = new List<String>();
List<?> l2 = ListUtils.clone(l);
```

Fig. 2. Capture conversion in method calls.

3.5. Type inference in method calls

In the context of reification of generic types, a technique should be provided so that type parameters inferred by the compiler for a generic method call are properly reified. As a simple example consider generic method `doClone()` in Fig. 2: at the call side, its type variable `T` can either be explicitly specified as in expression `<String>doClone(..)`, or be inferred as in expression `doClone(new List<String>(...))` (where it is inferred to `String`). In both cases, the actual instantiation of `T` should be reified and associated to the receiver code, so that instruction `new List<T>()` in method `doClone()` actually creates an object whose type is the same as argument `l`.

This requirement can be a problem since there are situations in which the compiler can actually infer types that are not *expressible types* of the Java language itself—types that a programmer cannot write down for they are not part of the actual Java language. Rather, it can be a non-expressible type [10], internally handled by the compiler.

As an example consider the following class declarations:

```
interface I1 { ... }
interface I2 { ... }
class A implements I1, I2 { ... }
class B implements I1, I2 { ... }
public class C {
   static <Z> Z choose(Z th, Z that) { ... }
   void main(){
      A z = choose(new A(), new B());
   }
}
```

The method `choose()` is called by passing as arguments an object of type `A` and an object of type `B`, respectively. In this situation the inference process leads to a *compound type* (namely, an *intersection* type [10]) `Object&I1&I2`. This is indicated by the fact that the above code raises the following compilation error

```
incompatible types
found   : Object&I1&I2
required: A
```

since type `A` for variable `z` does not match with the expected return type.

This problem of generic reification gets further complicated as far as wildcards are concerned. Wildcards allow to factor over different instantiation of the same generic class, as such, they have been fruitfully exploited for improving `javac` type inference in generic method calls [10,30]—this can be regarded as one of the most remarkable properties of Java wildcards. Unfortunately, such a flexibility comes at a price, as the following example reveals:

```
class A<X>{}
class B extends A<B>{}
class C extends A<C>{}
class D{
   static <Z> Z choose(Z th, Z that) { ... }
   void main(){
      choose(new List<B>(), new List<C>());
   }
}
```

This time, the method `choose()` is called by passing as arguments a `List<B>` and a `List<C>`, respectively. In this situation the inference process would lead to an infinite type, namely `List<? extends A<? extends A<? extends A<...>>>>`. However, since `javac` does not support infinite types yet [23], an approximation is used in which the infinite recursion is truncated by an unbounded wildcard as in `List<? extends A<? extends A<?>>>`. In a generic call of the kind

```
 choose(new List<Double>(), new List<String>());
```

it would even be inferred a type that is both compound and infinite. In a context of reification of generic types, approximating such types might lead to a mismatch between the compile-time and the runtime type of a given type variable. In order to avoid this inconvenience it would be possible to either (i) emit a warning when a non-expressible type is inferred for a method type variable or (ii) provide explicit support for both compound types and infinite types at runtime. Note that, by choosing the latter, we would have runtime types in which one or more arguments can be an infinite type: such a choice would hence require great care and investigation.

Requirement 5. The reification support should:

- represent generic types where the instantiation of a type parameter is the result of type inference.

3.6. Captured calls

In order to improve interoperability between wildcards and generic methods, Java allows to invoke a generic method passing as argument a wildcard type, as in the example shown in Fig. 2. Method `doClone()` is called within `clone()` with an argument of type `List<?>`, that is, without specifying an actual instantiation for type variable `T`—nor any true "type" can be inferred for it. Instead, as described in the JLS [10], the compiler assumes that type variable `T` is instantiated to type notation `?` (more precisely to the type `Z`, where `Z` is a fresh type-variable that has been generated during capture-conversion), meaning "any type"; hence the above code is correctly compiled, and when executed it assigns to variable `l2` a `List` object. We call this kind of invocation *captured call*.

In the context of reification, however, the semantics of method `doClone()` – that is, creating a new object with same content and type of the argument passed – can be preserved only if there is a way to consider as instantiation for `T` a concrete Java type (say `String`) instead of `?`. This would guarantee to correctly create the object through expression "`new List<T>()`", that is, to assign a `List<String>` object to `l2`. Put in general terms, captured calls can correctly support runtime types only if there is a way of dynamically recovering the right instantiation of the method type parameters from the runtime type of the arguments passed. Assuming that $<X_1, X_2, \ldots X_n>m(F_1, F_2, \ldots F_k)$ is a generic method declaration and $m(t_1, t_2, \ldots t_k)$ a generic method call, where the runtime types of $t_1, t_2, \ldots t_n$ are $T_1, T_2, \ldots T_k$, respectively, the following steps are required to infer concrete runtime types for each type parameter X_i in `m`:

(1) Consider the type U_i that has been inferred (at compile-time) for the type variable X_i of `m`. It can be either:
   - a resolved type `C`—in this case no further step is required
   - a fresh type variable—in this case a concrete type is found using the rest of the arguments (see below).

(2) the list of types $F_1, F_2, \ldots F_k$ is searched in order to find the formal argument F_j containing X_i as a top-level type argument (say in position *p*).

(3) The type corresponding to the instantiation of X_i is hence the p_{th} actual type parameter of T_j.

As stated in [10,30], a captured type variable X_i may only occur as a top-level type argument within the declaration of a given generic method `m`. Moreover the compiler disallows a captured type variable to appear in more than one formal arguments of `m`. Those assumptions ensure that a path to the concrete type that has to be inferred for X_i exists and is unique.

Requirement 6. The reification support should:

- in a generic method, be able to extract the actual instantiation of a type variable from the actual method's arguments;
- make such an instantiation available during method execution.

4. The EGO compiler

Soon after GJ was proposed for implementing generics in Java [19], and early compile-time proposals for reification had no great success [4,34,18], it was commonly accepted that an effective and efficient reification support for generics could be supported only by a change to the run-time system, as proposed e.g. in [1,22,14,13,2] and implemented in .NET. The research leading to the EGO compiler, culminating in the present article, shows instead that even a compile-time approach can be exploited similarly to other existing Java language features such as e.g. inner classes—though solutions at the JVM level have obviously better performance and coherence, and may be more appealing for an official release of the language.

```
class SimpleList<X> {

   X head;
   SimpleList<X> tail;

   SimpleList(){}

   static <Z> SimpleList<Z> make() {
      return new SimpleList<Z>();
   }

   public static void main(String[] args) {
      Object o = new SimpleList<String>();
      SimpleList<Integer> li = SimpleList.<Integer>make();
      boolean res = o instanceof SimpleList<String>;
   }
}
```

Fig. 3. An example of generic source code.

The EGO compiler (Exact Generics on-Demand) [33] is the result of a project developed in collaboration with Sun Microsystems with the goal of evaluating a smooth support to runtime generics, which would not require changes on the JVM or on any other component of the Java Runtime Environment (JRE). The solution conceived and developed is a sophisticated translation of code based on the type-passing style [34,32] – also known as lifting of type parameters – where runtime type information is automatically created on a by-need basis, and cached for future utilisation.

The main idea of EGO's translation scheme is to reify the generic type used to create an object to an actual further argument (called *descriptor*) to be passed to the constructor. The constructor is then automatically modified so that this descriptor gets automatically stored into a newly-generated field for later accesses: in this way each instance of a generic class will hold a reference to its exact generic type. Such an information is later used when necessary, e.g. when a cast operation occurs, when executing a type test, or when serialising the object. Several critical issues had to be tackled in order to make this general idea a fully-fledged solution, including performance, compatibility, and so on. In particular, EGO compiler has been developed with the following features: (*Laziness*) Descriptors are created only the first time they are required, preventing any interference with usual Java class loading dynamics, and avoiding the problem of infinite polymorphic recursion [34]; (*Completeness*) The type-passing translation schema is applied not only to generic classes, but also generic methods, generic inner classes, interfaces, and arrays; (*Effectiveness*) A number of bridging techniques were introduced to deal with effectiveness issues such as interoperability between legacy and generic Java code and support to separate compilation; (*Efficiency*) The need to obtain good performance results of the translated code pervasively affected all the aspects of the translation; and (*Implementation modularity*) EGO compiler is implemented as a modular extension to the `javac` compiler, namely, a further translation step to the abstract syntax tree (AST) just before the type-erasure process.

In this section we provide a brief overview of the translation schema of EGO, which in next section will be extended to deal with wildcards.

4.1. Type descriptors in EGO

EGO compiler represents generic types information by so called *class descriptors*. The class `Cla` used for representing the class descriptor for a type `C<T>` is structured in three main parts: (i) a `Class` object which stands for the class `C`; (ii) an array of descriptors which keeps track of the instantiation of type parameters, in this case containing only `T`'s type descriptor; and (iii) a reference to the descriptor for `C<T>`'s direct supertype. As already mentioned, arrays are handled through a special case of class descriptor, namely, a class `Arr` extending from `Cla`. Accordingly, the below discussion focuses on generic classes, while it smoothly applies to arrays as well. Similarly, other abstractions like raw types, generic inner classes and generic interfaces, are implemented through proper kinds of descriptors, which are not discussed in detail for the sake of simplicity—e.g. the raw type for `List` is represented by a type descriptor `List<Any>` where `Any` is a special descriptor.

EGO provides a hash-consing mechanism to quickly store and retrieve descriptors [26]. When a descriptor is required it is first searched in a global *descriptor registry* (a hashtable): if it is not found the descriptor is created and registered there. Moreover, the reference to a descriptor is also stored locally to where it has been used, e.g. as static field of a client class, leading to a particularly space- and time-efficient double-caching mechanism [33]. The details of this kind of management are encapsulated into a method `$crCLA()`, automatically added by the EGO compiler to a generic class (`C`), which takes the instantiation of the type parameters (`T`) and yields the corresponding descriptor of the current class (`C<T>`), also taking care of other tasks such as registering the descriptor and setting the descriptor for its direct supertype.

```
class SimpleList<X> implements EGO.Parametric {
  protected Desc.Cla $d; // Instance descriptor
  static Desc[] $descs = new Desc[5]; //Local descriptor cache
  X head;
  SimpleList<X> tail;
  // Constructor (for backward compatibility)
  SimpleList() {
    this((Desc.Cla)$C(0));
  }
  SimpleList(Desc.Cla $d) {
    this.$d = $d;
  }
  static <Z> SimpleList<Z> make(Desc.Meth $md) {
    return new SimpleList<Z>($B$D($md,0));
  }
  public static void main(String[] args) {
    Object o = new SimpleList<String>($C(2));
    SimpleList<Integer> li = SimpleList.<Integer>make($C(4));
    boolean res = $C(2).isInstance(o);
  }
  // Facility method to register descriptors
  public static Cla $crCLA(Cla[] params) {
    Cla $v = Cla.reg(SimpleList.class, new Cla[]{params[0]});
    $v.setFath(Desc._Object);
    return $v;
  }
  // Facility method for retrieving closed descriptors
  private static Desc $C(int id) {
    if ($descs[id] != null) return $descs[id];
    switch (id) {
       case 0: return $descs[id] = $crCLA(new Cla[]{Desc._Any});
       case 1: return $descs[id] = Cla.reg(String.class);
       case 2: return $descs[id] = $crCLA(new Cla[]{$C(1));
       case 3: return $descs[id] = Cla.reg(Integer.class);
       case 4: return $descs[id] = Meth.reg("make",new Cla[]{$C(3});
     }
     return null;
  }
  // Facility method for retrieving open descriptors
  private static Desc $B$D(Desc d, int id) { ... }
}
```

Fig. 4. Translation with EGO of code in Figure 3.

4.2. Type passing Technique in EGO

Fig. 3 reports an example of class `SimpleList<X>`, and Fig. 4 its corresponding translation. An argument of type `Cla` is added to the constructor, representing the generic type under instantiation. Its content will be stored in the EGO-generated field `$d`: this is meant to contain information about the runtime type of the current instance, passed from the client that invokes the constructor. Note that the legacy constructor is kept to support compatibility with legacy code: there, the new constructor is called by passing a special descriptor for the raw type of `SimpleList`, namely, the type given by the type system to the non-generic type `SimpleList`. Moreover, it is common practice in Java to add synthetic fields and methods during compilation – e.g. bridge methods for generics [19], or new fields for inner classes – so introspection by Reflection would be supported by a similar compatibility degree. The reification schema when creating an object is of the general kind:

```
new List<X>(e1,e2,..) -> new List<X>(/*Desc for List<X>*/,e1,e2,..)
```

namely, an appropriate expression – which is in charge of efficiently creating/retrieving the descriptor – is added as first argument of a generic class' constructor. Descriptors can be of two different kinds: (i) they can be independent of the current generic instantiation, such as e.g. type `List<String>`, which we call *closed descriptors*, or (ii) they may include type variables of the scope, such as `List<Z>` in method `List.<Z>make()`, which we call *open descriptors*. These two kinds of descriptor require different management, as shown in [33], delegated, respectively, to methods `$C()` and `$B$D()` as shown in Fig. 4: independently of their details, these methods are in charge of implementing the first caching level. For instance, method `$C()` looks first for the required descriptor in static field `$descs`, otherwise a new descriptor is created and registered through

```
class Cla extends Desc {
   Class<?> theClass;
   Desc[] params;
   int[] annotations;
   Cla[] bounds;
   Cla super;
}
```

Fig. 5. Structure of a class descriptor for wildcards.

method `$crCLA()`. Generic methods are handled similarly as shown for `<Z>make()`: a method descriptor (instance of class `Desc.Meth`) is passed as first argument in the invocation, carrying information about the instantiation of the method type parameters—independent of it being inferred or explicitly provided.

A type-dependent operation involving a generic type exploits the runtime type information stored in the `$d` field. For instance, let `v` stand for the expression used to access the descriptor for `List<String>` – e.g. `$C(2)` as in method `main()` –, we have the translations:

```
o instanceof List<String> -> v.isInstance(o)
(List<String>)o -> (List<String>)v.cast(o)
```

Methods `isInstance()` and `cast()` (of class `Cla`) simply try to access `o`'s descriptor: if this is possible it means the objects has been created from a generic class, hence they simply check whether such a descriptor corresponds to a descriptor for any supertype of `List<String>`—by properly implementing the subtyping test. Other kinds of runtime introspection, such as e.g. those required to support persistence, are implemented in a similar fashion.

Although from Fig. 3 it might appear that the translation dramatically changes the shape of the source code, it actually mostly involves only the creation of few and small synthetic methods and fields, and the insertion of access expressions for descriptors. Performance measures executed over large-size benchmarks, like the `javac` compiler itself, report a general time overhead within 10%, memory overhead within 5% and a class-size overhead within 15% [33].

The translation is affected by many other aspects which are not discussed here but are presented in greater detail in [33], the description we provided touches the parts that are relevant for understanding the extension implemented for supporting wildcards.

5. Wildcards in EGO compiler

Based on the issues identified in Section 3, we here describe how the EGO compiler has been extended to deal with wildcards, leading to a complete reification proposal for Java 5.0/6.0 language.

5.1. Runtime representation of wildcards

The first issue to be addressed concerns the representation of WTs. Among the two possibilities identified in Section 3.1 we here adopts the VPT-like representation, which is smoothly applied to EGO. In particular, it suffices to add to each type argument an annotation value, expressing which wildcard symbol is used in that position, if any. The class for descriptors `Desc.Cla` (whose structure is shown in Fig. 5) has been modified so that a new field `annotations` now keeps track of the annotation symbol associated to each type parameter, each mapping into an integer value. In particular, 0 means an invariant argument (no wildcard), 1 means covariant argument (wildcard "`? extends`"), 2 means contravariant argument (wildcard "`? super`"), and 3 means bivariant argument (wildcard "`?`"). This field is typically left `null` if the class descriptor represents an invariant type, that is, if all its arguments have no wildcard. Moreover, field `bounds` has been added as well which keeps track of the bounds of each type variable. For instance, EGO represents the type `SimpleList<? extends String>` by the class descriptor where: (i) field `theClass` points to the `Class` object of `SimpleList`; (ii) `params` contains one element, that is, the type descriptor for `String`; (iii) `annotations` contains one integer element, whose value is 1; (iv) `bounds` contains one descriptor for the type variable bound, namely `Object`; (v) the `super` field refers to the type descriptors of the supertype of `SimpleList<String>`, namely, `Object`.

Concerning the association between objects and the runtime representation of their types, even with WTs this extension exploits the standard EGO paradigm: namely, each object carries in its compiler-generated field `$d` the descriptor for its type, to be used in type-dependent operations.

5.2. Capture conversion and subtyping in EGO

In order to correctly implement subtyping, we first have to change the implementation of method `$crCla()` shown in Fig. 3, which has the goal of creating and registering descriptors of a given class. Fig. 6 shows an example. First, the method now has a further argument containing the annotations array, which is required when creating WT descriptors. For instance, to create the descriptor for `D<? extends String>`, an array with one descriptor for `String` is to be passed as first argument,

```
class D<X> extends C<C<? extends X>>{
   ...
}
```

```
class D<X> extends C<C<? extends X>> {
 ...
 public static Cla $crCLA(Cla[] params, int[] annotations) {
    Cla $v = Desc.Cla.reg(D.class, params, annotations);
    $v.setTypeVarBounds(new Cla[]{Desc._Object});
    $k = Desc.capture($v);
    Cla $super=C.$crCLA(
                 new Cla[]{C.$crCLA(new Cla[]{$k.params[0]},
                 new int[]{1})});
    $v.setFath($super);
    return $v;
 }
}
```

Fig. 6. Method `$crCLA` in the case of wildcards.

while the array `new int[]{1}` is passed as second argument (1 stands for covariance). Inside the method body, there is a call to the `$capture` method of the `Cla` class, which actually implements capture conversion precisely as described in Section 3.2. The descriptor returned is then used for registering the descriptor for the direct supertype.

Of course, to support capture conversion at runtime we also need to support a new kind of descriptor, namely the fresh type variable (`Ftvar`) descriptor. A `Ftvar` descriptor represents a fresh type variable that has been created during the capture conversion process. Each `Ftvar` descriptor simply contains information about its uniqueness and the descriptors for its upper bound and lower bound. Capture conversion can thus be seen as an operation that takes as input a (possibly generic) class descriptor $\texttt{D}_{\texttt{in}}$ and whose output is another class descriptor $\texttt{D}_{\texttt{out}}$, where type parameters with annotation symbol 1, 2 or 3 are turned into new `Ftvar` descriptors.

Now that each descriptor properly points to its supertype descriptor, the subtyping algorithm is simply implemented by navigating the inheritance chain (step 1 of the subtyping algorithm), and then proceeding recursively on type arguments, as described in Section 3.3. The code in Fig. 7 shows how the subtyping algorithm supporting WTs can be implemented leveraging EGO type descriptors.

Since no general solution has been found so far for intercepting non-terminating computations, in EGO we currently set as 100 the maximum number of calls caused by a single subtyping test. Would any such solution be found, our framework appears flexible enough to easily incorporate it.

5.3. Type inference and captured calls

Our reification of WTs provides the necessary means to support capture calls, following the details described in Section 3.6. Fig. 8 shows details of the translation of the `ListUtils` class shown in Fig. 2. Given a captured call to a method `m` with *n* type variables, we can look at the captured call inference process as a function that takes as input the runtime type of arguments passed to `m` and returns as output the *n* inferred types for the method's type variables. This is obtained in EGO compiler by a method `$KAP`, added to all the client classes that perform a captured call, and which encapsulates the above inference process: it takes the descriptors of all the arguments passed and returns the method descriptor to be actually used in the invocation. Hence, invocation `doClone(l)` is translated to `doClone($KAP(l.$getDesc()),l)`. Assuming that `l` is an object whose runtime type is `List<String>`, `$KAP` takes in input the descriptor for `List<String>`, matches it with the original signature `<T>clone(List<T>)` thus binding `T` to `String`, and returns the descriptor for method `ListClient.<String>clone()`.

The internal implementation of `$KAP` is as follows. The static field `$kap0_path` is added that encodes the position of the type variable to be inferred inside the argument type—if more arguments exist, an array of such fields has to be added. Note that, since a captured type variable can only appear at the toplevel of a given generic type `C<T>` [10,29,30], an integer value suffices to encode such information. In this case the field `$kap0_path` is given the value `0`, meaning the first type argument is already the variable to be inferred. If the argument to method `clone()` were e.g. `Pair<String,T>`, that field would have been set with the value `1`, namely, `T` is found there moving to type argument in position 1 of `Pair<String,T>`. After `$kap0_path` is initialised, method `Desc.infer()` navigates the type finding the type parameter to be inferred, hence a method descriptor is correspondingly created and returned.

This implementation of `Desc.infer()` actually might have to handle some other subtleties which we do not fully describe here for the sake of simplicity, including `null` values passed as argument, actual argument types that are proper subtypes of formal ones, and interplay between captured calls and open descriptors. The basic translation schema is anyway similar.

```
public boolean isSubtypeOf(Cla that) {
   Cla cla = this;
   // Navigating the inheritance chain
   while(cla.theClass != that.theClass && !cla.equals(ObjDesc) {
      cla = cla.super;
   }
   if (cla.equals(ObjDesc)) {
      return that.equals(ObjDesc);
   }
   // Capturing the two types
   Cla s = cla.capture();
   Cla t = that.capture();
   for (int i=0;i<params.length;i++) {
      Desc ui = s.params[i];
      Desc vi = t.params[i];
      // Checking interval containment
      if (vi instanceof Ftvar) {
         Desc lower = vi.getLowerBound().subst(vi,ui);
         Desc upper = vi.getUpperBound().subst(vi,ui);
         if (!lower.isSubtypeOf(ui) || !ui.isSubtypeOf(upper)){
               return false;
         }
      }
      else if (!ui.equals(vi)){
         return false;
      }
   }
   return true;
}
```

Fig. 7. The subtyping algorithm.

```
class ListUtils {
   ...
   public static List<?> clone(List<?> l){
      return doClone($KAP(l.$getDesc()),l);
   }
   ...
   protected static int $kap0_path = 0;
   private static Met $KAP(Cla $cd){
      return Met.reg((Cla)$C(0),
                     new Cla[]{Desc.infer($cd, $kap0_path)});
   }

   private static Desc $C(int id) {
      ...
      //$C(0) builds a descriptor for ListClient
   }
}

class Desc {
...
   public static Cla infer(Cla cla, int path) {
      return cla.params[path];
   }
}
```

Fig. 8. Translation of code with a captured call.

6. Discussion and conclusions

In this paper we discussed all the issues that must be tackled to implement a reification support for wildcards, to be used in a new or existing approach for runtime generics, either at compile-time or runtime. The main achievement of this work was to pick the necessary details on existing scientific articles, the Java Language Specification, and the reference implementation of `javac` compiler, and accordingly fill the gap towards a complete implementation of wildcard support.

Our techniques have been successfully implemented in an extension to the EGO compiler. This resulted in a reification proposal for generics that, to the best of our knowledge, is the only one entirely dealing with the Java Programming Language of versions 5.0/6.0. It should be noted that a reification support is basically a runtime system for a new language, namely, a slight extension of Java where generics (and wildcards) are treated as first-class types—seamlessly usable in type-dependent operations. Such a language is currently not deployed, hence it is very difficult to gather large-size source code upon which performing correctness/performance tests. Other than small-size synthetic programs, our reference case was the code for `javac` itself, which largely relies on generics and performs some legacy-style type conversions to unbounded generics such as `C<?>` [5]. From that experience, we were able to observe the correctness of our approach with respect to the results of the compilation process, and a zero overhead with respect to the performance of EGO before supporting wildcards. This result has been obtained despite some operations involving wildcards are intrinsically more complex to execute – up to one order of magnitude – compared to standard ones, like subtyping and capture calls. An important point is that, however, the occurrences of type-dependent operations involving wildcards will likely be quite infrequent—indeed, although their reification is necessary, wildcards remain a mainly static mechanism for type-safety.

The inclusion of wildcards after Java 5.0 is a challenging requirement for existing works on reification. A main reason is that most features of WTs – e.g. subtyping – are not easily mimicked by standard (non-generic) Java, and hence they might be difficult to implement through smooth extensions of the legacy Java compiler or VM. NextGen [4] for instance is conceived around the idea of reusing concrete Java classes to simulate each different instantiation of a generic type used in an application—class `List$String` for type `List<String>` and so on. Such an approach, which could be implemented either as a compiler or as an extended class-loader [20], can lead to serious implementation issues. Concerning subtyping for instance, types of the kind `List<? super T>` are contravariant, hence, the set of their supertypes is not closed: for any newly defined class `C` such that `C<:T`, type `List<? super C>` should be a supertype of `List<? super T>`. Therefore, whether such approaches would ever be able to support subtyping is an open issue.

An issue we discussed in this paper which is studied elsewhere is decidability of subtyping with variance. In [11] it is described an extension of the C# language featuring variance annotation in parameterised class declarations (declaration-site variance). Since the .NET framework provides runtime support for generic types, some of the problems tackled in this paper have been already discussed in [12]—in particular those regarding subtyping and decidability. One obvious direction for future work is to give a better characterisation of the decidability issues addressed in this paper, in particular investigating which restrictions (if any) should be applied to the language in order to make subtyping decidable. Moreover, the problem of dealing with inexpressible types inferred by `javac` in generic method calls has not been explored in depth. In particular, it is not clear (i) how those types could be used within instance creation expressions and (ii) whether providing runtime support for those types would result in further complicating the decidability issue.

References

[1] O. Agesen, S. Freund, J.C. Mitchell, Adding type parameterization to the Java language, in: Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM, New York, Atlanta, Georgia, 1997.
[2] S. Alagić, M. Royer, Genericity in Java: persistent and database systems implications, The VLDB Journal The International Journal on Very Large Data Bases 17 (4) (2008) 847–878.
[3] N. Cameron, E. Ernst, S. Drossopoulou, Towards an existential types model for Java wildcards, in: Formal Techniques for Java-like Programs, FTfJP 2007, 2007. URL: http://pubs.doc.ic.ac.uk/towards-existential-wildcards/.
[4] C. Cartwright, G. Steele, Compatible genericity with run-time types for the Java programming language, in: Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM, New York, Vancouver, Canada, 1998.
[5] M. Cimadamore, M. Viroli, Reifying wildcards in Java using the EGO approach, in: Proceedings of the 2007 ACM Symposium on Applied Computing, SAC 2007, ACM, Seoul, Korea, 2007 (special Track on Programming Languages).
[6] A. Igarashi, B.C. Pierce, P. Wadler, Featherweight Java: A minimal core calculus for Java and GJ, ACM Transactions on Programming Languages and Systems 23 (2001) 396–450.
[7] A. Igarashi, B.C. Pierce, P. Wadler, A recipe for raw types, in: 8th Workshop on Foundations of Object-Oriented Languages, University of Pennsylvania, Philadelphia, PA, London, England, 20 January, 2001. Web site: http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL8.html.
[8] A. Igarashi, M. Viroli, On variance-based subtyping for parametric types, in: European Conference on Object-Oriented Programming, ECOOP 2002, in: LNCS, vol. 2347, Springer-Verlag, 2002.
[9] A. Igarashi, M. Viroli, Variant parametric types: A flexible subtyping scheme for generics, ACM Transactions on Programming Languages and Systems 28 (5) (2006) 795–847.
[10] B. Joy, J. Gosling, G. Steele, G. Bracha, The Java Language Specification, third ed., Addison-Wesley, New York, 2005.
[11] A. Kennedy, C. Russo, B. Emir, D. Yu, Variance and generalized constraints for C# generics, in: European Conference on Object-Oriented Programming, ECOOP, in: LNCS, vol. 4067, Springer-Verlag, 2006.
[12] A.J. Kennedy, B.C. Pierce, On decidability of nominal subtyping with variance, in: FOOL-WOOD, 2007.
[13] M. Day, R. Gruber, B. Liskov, A. Myers, Subtypes vs. where clauses: Constraining parametric polymorphism, in: Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM, New York, Austin, TX, 1995.
[14] A.C. Meyers, J.A. Bank, B. Liskov, Parameterized types for Java, in: Symposium on Principles of Programming Languages, ACM, New York, Paris, France, 1997.
[15] Microsoft, C# language specification 2.0, Tech. Rep., Microsoft, 2003. Web site: http://msdn.microsoft.com/vcsharp.
[16] J.C. Mitchell, G.D. Plotkin, Abstract types have existential types, ACM Transactions on Programming Languages and System 10 (3) (1988) 470–502.
[17] M. Odersky, The Scala language specification, Version 2.6, 2007. http://www.scala-lang.org.
[18] M. Odersky, E. Runne, P. Wadler, Two ways to bake your pizza — Translating parameterized types into Java, CIS- 97-016, University of South Australia, Adelaide, Australia, 1997.
[19] M. Odersky, P. Wadler, G. Bracha, D. Stoutamire, Making the future safe for the past: Adding genericity to the Java programming language, in: Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM, New York, 1998.

[20] J. Sasitorn, R. Cartwright, Efficient first-class generics on stock Java virtual machines, in: H. Haddad (Ed.), Proceedings of the 2006 ACM Symposium on Applied Computing, SAC, Dijon, France, April 23–27, 2006, ACM, 2006.
[21] Slashdot site, Preview of Java 1.5. URL: http://developers.slashdot.org/article.pl?sid=03/05/30/1942259.
[22] J.H. Solorzano, S. Alagic, Parametric polymorphism for Java: A reflective solution, in: Conference on Object-Oriented Programming, Systems, Languages and Applications, ACM, New York, Vancouver, British Columbia, Canada, 1998.
[23] Sun Microsystems, Bug 4929881. URL: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4993221.
[24] Sun Microsystems, Bug 6207386. URL: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6207386.
[25] Sun Microsystems, Bug 6558545. URL: http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6558545.
[26] D. Syme, A. Kennedy, Design and implementation of generics for the.NET common language runtime, in: Programming Languages Design and Implementation, ACM, New York, Snowbird, Utah, 2001.
[27] K.K. Thorup, M. Torgersen, Unifying genericity — combining the benefits of virtual types and parameterized types, in: European Conference on Object-Oriented Programming, in: LNCS, vol. 1628, Springer-Verlag, Berlin, Lisbon, Portugal, 1999.
[28] M. Torgersen, The expression problem revisited: Four solutions using generics, in: European Conference on Object-Oriented Programming, ECOOP'2004, in: LNCS, vol. 1445, Springer, Oslo, Norway, 2004.
[29] M. Torgersen, E. Ernst, C. Plesner Hansen, F.J. Wild, in: P. Wadler (ed.), Proceedings of FOOL 12, ACM, School of Informatics, University of Edinburgh, Long Beach, CA, USA, 2005. Electronic publication. URL: http://homepages.inf.ed.ac.uk/wadler/fool/.
[30] M. Torgersen, C. Plesner Hansen, P. von der Ahé, E. Ernst, G. Bracha, N. Gafter, Adding wildcards to the Java programming language, Journal of Object Technology 11 (3) (2004) 1–20.
[31] M. Viroli, On the recursive generation of generic types, Technical Report DEIS-LIA-00-002, LIA(42), Alma Mater Studiorum - Università di Bologna, 2002.
[32] M. Viroli, A type-passing approach for the implementation of parametric methods in Java, The Computer Journal 46 (3) (2003).
[33] M. Viroli, Effective and efficient compilation of run-time generics in Java, in: V. Bono, M. Bugliesi, S. Drossopoulou (Eds.), 2nd Workshop on Object-Oriented Developments, WOOD 2004, in: Electronic Notes in Theoretical Computer Science, vol. 138(2), Elsevier Science B.V., 2005, pp. 95–116, CONCUR 2004, London, UK.
[34] M. Viroli, A. Natali, Parametric polymorphism in Java: An approach to translation based on reflective features, ACM SIGPLAN 35 (10) (2000) 146–165. Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2000, Minneapolis, MA, USA, 15–19 October 2000.
[35] M. Viroli, G. Rimassa, On access restriction with Java wildcards, in: OOPS Track at ACM SAC, Journal of Object Technology 4 (10) (2005) (special issue).
[36] D. Yu, A. Kennedy, D. Syme, Formalization of generics for the.NET common language runtime, in: N.D. Jones, X. Leroy (Eds.), Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, 14–16 January, 2004, ACM, 2004.