Università degli Studi di Bologna
DEIS

Parametric Polymorphism in Java through the Homogeneous Translation LM: Gathering Type Descriptors at Load-Time

Mirko Viroli   Antonio Natali

April 1,2000

DEIS Technical Report no. DEIS-LIA-00-001   *LIA Series no. 41*

Parametric Polymorphism in Java through the Homogeneous Translation LM: Gathering Type Descriptors at Load-Time

Mirko Viroli Antonio Natali

DEIS Department
University of Bologna
Viale Risorgimento, 2 Bologna - Italy
`mviroli,anatali@deis.unibo.it`

April 1,2000

Abstract. The introduction of parametric polymorphism in Java with translation approaches has been shown to be of considerable interest, allowing the definition of extensions of Java on top of the existing Virtual Machines. Homogeneous translations furthermore, seem to be more useful than heterogeneous, avoiding the continuous increase of library code with redundant information. At this time however, homogeneous approaches aren't as flexible as heterogeneous, with extensions failing to integrate well with base language typing. In this paper, using some of the features of the Core Reflection of Java, we introduce a homogeneous translation in which run-time information about instantiation of type-parameters is carried, allowing full integration of parameterized types with Java typing. Performance overhead is highly decreased using a brand new translation technique based on the deferring of the management of type information at load-time. The same power and flexibility of previous heterogeneous approaches is obtained while maintaining homogeneous translation advantages.

Keywords: *Java, Parametric Polymorphism, Compilators, Translators*

Contents

1 Introduction

Software Engineering practice suggests the writing of generic code, which by capturing a common behaviour, can be used in different contexts to achieve a high degree of reuse. Java allows this with inheritance, but lacks the mechanism of genericity, called parametric polymorphism, that allows programmers to abstract classes with respect to types.

In the last few years there has been a lot of research done on extensions of Java with parameterized types. In [SA98] evaluation criteria of extension features and implementations are discussed. Java language extensions can mainly be realized in two ways: by redefining the run-time system of Java or by defining translators from the extended language to standard Java.

The first approach can be seen in Meyer, Bank and Liskov's PolyJ work ([MBL97]), where a parametric polymorphism with *where* clauses that constrains the instantiation of type variables is studied. In Agesen, Freund and Mitchell's work ([AFM97]) instead, the class file format is extended and the class loader redefined, allowing the management of parameterized types. Extending the Java architecture leads to more flexibility and power, but obviously loses in compatibility with Java standard platforms.

The second approach, initially used in Odersky and Wadler's *Pizza* ([OW97]) however, produces classes that work with old libraries and on all present Java platforms. There are two kinds of approaches to translation: homogeneous and heterogeneous. Homogeneous translations, whose prototype is the one described in Odersky, Wadler, Bracha and Stoutamire's *GJ* ([OW98]), create one *erased* class for each parameterized class, in which the type variables are substituted with the Java super-type `Object`. For instance an array of `Object` elements is created when a generic array is required, and down-casts conversions (narrowing) allow the retrieval of correctly typed elements. *GJ* has also been studied from a theoretical point of view in [APW99], in which its key features are formalized using a core calculus called *Featherweight Java*, which has been proved to be type-safe.

In heterogeneous translations instead, different instantiations of type parameters cause different classes to be created, as occurs with early implementations of templates in *C++*. *Pizza* may be implemented with both approaches, while Cartwright and Steele's *NextGen* ([CS98]) can be considered as following the heterogeneous approach, even if it actually uses a hybrid one, where one erased class for each parameterized class and one light wrapper class and one interface for each instantiation are created.

Another way to add genericity to Java is through virtual types, which was proposed in [Tho97], [Tor98] and [BOW98].

Here we focus on translations, and in particular on the homogeneous approach, that avoids the possible creation of a large number of class files with redundant information due to the different instantiation of parameterized classes. At the moment, heterogeneous translation is the only translation approach that is able to completely integrate parameterized types with Java typing. This means that each parameterized type and type variable can be used in every context where a type identifier is expected. Operations such as casts and instance tests are appliable to generic types only if run-time information about their actual instantiation is carried, or known at run-time, as in heterogeneous translation.

This missing step is covered by introducing a homogeneous translation in which parameterized classes carry the instantiations of their parameters, by using the run-time class representation feature given by the core Java Reflection, that of class `java.lang.Class`. In this framework the Java Core Reflection ([JCR]) can be used to perform type-dependent operations such as instantiations, casts and instance tests over both type variables and parameterized types, allowing full compatibility between generic types and Java types.

The straightforward implementation of such a translation leads to relevant performance drawbacks, for this reason it has never been proposed as a real solution for adding genericity to Java. The main contribution of this paper is the idea of moving the overhead needed to manage such type information from run-time to load-time. This leads to a new translation, called *LM* (Load-Time Management), which solves the main drawbacks of previous approaches, namely the lack of integration of *GJ* and the continuous creation of redundant class files of *NextGen*, while keeping performance overhead bounded to acceptable values.

A further contribution of this paper is to highlight the possible definition of an infinite number of types within a program, which may be caused by two or more parametric classes mutually referring to each

```
class Cell<T>{
    private T t;
    Cell(T t){ this.t=t; }
    void set(T t){ this.t=t;}
    T get(){ return t;}
    void setIfPossible(Object o){
        if (o instanceof T) this.t=(T)o;
    }
    Cell<String> getCString(){
        return new Cell<String>(""+t);
    }
}
```

```
class Pair<R,S> extends Cell<R>{
    private S s;
    Pair(R r,S s){
        super(r);
        this.s=s;
    }
    void setSecond(S s){ this=s; }
    S getSecond(){ return s;}
    Pair<S,R> getReverse(){
        return new Pair<S,R>(s,get());
    }
}
```

```
class Client{
    public static void main(String[] s){
        Cell<Integer> c1=new Cell<Integer>(null);
        c1.set(new Integer(1));
        Integer I=c1.get();
        c1.setIfPossible("2"); // "2" is not stored into c1
        Cell<String> c2=c1.getCString(); // Cell<String>("1")
        Pair<Integer,String> p=new Pair<Integer,String>(I,c2.get()); // pair[1,"1"]
        Object o=p.getReverse();
        c1=(o instanceof Cell<Integer>)?(Cell<Integer>)o:null; // c1=null
    }
}
```

Figure 1. *Source code*

other. In these cases implementations which statically gather information about the parametric types used, like the heterogeneous approaches (*Pizza* and *NextGen*) and our own, cause the translation of a program to fail to terminate. Here we give a necessary and sufficient condition for a program not to cause such inconvenience, allowing early detection.
The rest of the paper is organized as follows. Section 2 focuses on the problem of the full integration of parametric types into Java, by giving a brief overview of past translations. Section 3 depicts the general idea of carrying run-time information about the instantiation of type variables. Section 4 shows *LM* translation, while section 5 presents and discusses performance measures done over the translated code. Section 6 discusses the translation termination problem and section 7 completes section 4 by sketching the implementation of usual language features related to parametric types, such as parametric interfaces, parametric methods, and so on. Section 8 presents typical implementation issues about compatibility, security and persistence.

2 A brief survey of past translations

In order to explain the problem of the full integration of parametric types into Java, which motivated the study of *LM* translation, it helps to present a source code that uses some genericity features. We will use that code both to fix the terminology we will use, and later to give the corresponding translation using *LM*. See Figure 1.

Class `Cell<T>` is a simple wrapper for the type `T`, it is a parametric (or generic) class which abstracts from the type `T`, which is called its *type variable* (or its *type parameter*). Class `Pair<R,S>` extends from `Cell<R>`, its first type variable will always be the same as the type variable of its direct supertype. When a class uses some parametric types within its code in some operations, like class-instantiation (creation of objects) and definition of variables, then we say that the class is a *client* for these parametric types. This holds for instance for class `Client`, which uses the parametric classes `Cell<T>` and `Pair<R,S>`.
As one may notice clients do not use parametric classes before having assigned actual types to all their type parameters. The type one obtains with such a binding operation, is called an *instantiation* of the parametric type. So for example `Client` uses the type `Cell<Integer>`, which is an instantiatiation for the class `Cell<T>` in which type variable `T` is instantiated to the type `Integer`.

Class `Cell<T>` provides getter and setter methods for its field `t`, and the method `setIfPossible`, which sets `t` only if the input argument conforms to the current instantiation of the type variable `T`. Such a control is performed using the core Java type operation `instanceof`, which when dealing with genericity, should accept parametric types too. Finally `Cell<T>` also provides a method that creates and returns a new class instance (i.e. an object, through a `new` expression) of the parametric class instantiation `Cell<String>`. The class `Pair<R,S>` has methods to set and to get the second element of the pair, and a method which returns a `Pair` object whose instantiations of type variables are reversed with respect to those of the receiver `Pair` object.
Class `Client` uses parametric types to define variables, and to perform type operations such as class-instantiations, `instanceof` tests, and casts (type conversions). It also calls some method for these classes. Note that within the method bodies of the parametric classes some type-operation over their type variables may be realized, such as the `instanceof` test and the cast in the method `setIfPossible`.

2.1 GJ

One of the first approaches introduced to support parametric polymorphism in Java was *GJ* (Generic Java). Such a translator uses the erasure technique, that is it translates a parametric class, say `Vector<T>`, into the class a Java programmer would write to implement a vector in a generic way, i.e. a `Vector` which stores `Object` elements[1] . As `Object` is the root of the inheritance hierarchy of Java, this allows such vectors to be used to store any kind of object. This implementation of generic types is usually called the (homogeneous) *generic idiom*, and the erasure technique is the way a translator relies on it.
In order to be able to recover the actual signature of a parametric class from a class file at translation time, the signature should be added to the class file as a text, as if it were a comment. Being aware of such a possible change, the translator checks for the existence of the signature and is then able to manage in a appropriate way client classes. In particular the following main translations should be applied:

- Each access to a parametric class has to be translated into an access to the erased class
- When passing an object `O` as input argument of a method whose input type is a type variable `T`, the translator has to statically checks that the type of `O` is a subtype of the current instantiation of `T`.
- If the result type of a method is a type variable `T`, when the method is called the translator has to apply a downcoast to the returned object, in order to convert its type (which is `Object`) to the actual instantiation of `T`.

In Figure 2 is shown the translation of the source code of Figure 1.

2.2 The full integration problem

GJ does not handle any run-time carrying of type parameter instantiation, so it is unable to deal with type operations over type variables, or instance tests and cast operations over parametric types. This situation may cause the lack of a desirable integration of the programming mechanism introduced with the core language.
This well known problem, already mentioned in several works ([CS98][SA98][OW98][AFM97]), justified for example the introduction of the heterogeneous translation called *NextGen*, in which the run-time information on parametric types are obtained using a technique based on a sort of macro-expansion.

2.3 NextGen

The need of such an integration has anyway led to a particular kind of heterogeneous approach in which the redundant code is highly reduced. This is realized by using the erasure technique and filling the holes caused by the need of run-time type information using functionality provided by small wrapper classes,

[1]More generally, dealing with bounded polymorphism, a type variable is erased to its bound, as mentioned in Section 7.2

```
class Cell{
  private Object t;

  Cell(Object t){ this.t=t; }
  void set(Object t){ this.t=t;}
  Object get(){ return t;}
  void setIfPossible(Object o){
    if (o instanceof Object) //Illegal
        this.t=(Object)o; //Illegal
  }
  Cell getCString(){
    return new Cell(""+t);
  }
}
```

```
class Pair extends Cell{
  private Object s;
  Pair(Object r,Object s){
    super(r); this.s=s;
  }
  void setSecond(Object s){ this=s; }
  Object getSecond(){ return s;}
  Pair getReverse(){ return new Pair(s,get());
}
```

```
class Client{
    public static void main(String[] s){
        Cell c1=new Cell(null);
        c1.set(new Integer(1)); // Static control on the input argument
        Integer I=(Integer)c1.get(); // Cast applyed to the output result
        c1.setIfPossible("2"); // This functionality does not work
        Cell c2=c1.getCString();
        Pair p=new Pair(I,c2.get());
        Object o=p.getReverse();
        c1=o instanceof Cell? (Cell)o :null; //Illegal
    }
}
```

Figure 2. *GJ translation of classes Cell,Pair and Client*

built using the heterogeneous idiom. In particular NextGen creates one erased class for each parametric class as *GJ* does. Furthermore it also creates a light wrapper class and an interface for each instantiation of a parametric class. Such a wrapper class defines *snippet* methods used to perform type operations such as class-instantiation and array creation, while the interface is used to support subtyping, i.e. instance tests and casts. The main drawback of this approach is the production of redundant code as classes are instantiatied with different types.
A complication occurs when one tries to instantiate the type variable of a generic class using a private class of a package different than that of the generic class. In this case snippets are built into the client and passed to the wrapper class which needs them.
In Figure 3 we shows the correspondent translation, supposing that the class `String` would be private and in the same package as Client.

The class file overhead is reduced but it still remains, and it is not clear if such a framework could be actually applied to real systems.
In this paper we present an homogeneous translation which does not produce unnecessary redundant code, and which allows the carrying for the instantiations of type parameters at run-time, using the reflective features of Java. The language our translator will accept is the same as that of NextGen, it allows type operation over parametric types and type variables. Such a language is a strict superset of *GJ*.

3 Carrying Run-Time Information with Homogeneous Translations

To better describe *LM* translation, let us first point out a somewhat trivial yet interesting translation, which is the first solution one can think of when trying to reach the full integration using an homogeneous approach. We will call it *RM* translation (Run-Time Management Translation). The idea is very simple: each time a type operation over a given parametric type instantiation `A<T>` has to be realized, an object `t` storing all the needed information about the type `A<T>` is created, and the operation is translated into an appropriate method of `t`. Such objects will work as type descriptors. A possible implementation for their class is shown in Figure 4. When an object `O` belonging to a parametric class `A<T>` is created, a

```
class Cell{
  private Object t;

  Cell(Object t){ this.t=t; }
  void set(Object t){ this.t=t;}
  Object get(){ return t;}
  void setIfPossible(Object o){
    if ($snip$1(o)) this.t=o;
  }
  Cell getCString(){
    return new $$Cell$_String_$(""+t);
  }
  protected abstract boolean $snip$1(Object o);

  public abstract static class $IsolatedSnippets{
    boolean $1(Object o);
  }
}

interface $Cell_$String_$ {}

class $$Cell$_String_$ extends Cell
        implements $Cell_String_${
  static Cell.$IsolatedSnippets $is=null;
  public static void $init($IsolatedSnippets $is){
    this.$is=$is;
  }
  $$Cell$_String_$(Object t){ super(t);}
  protected boolean $snip$1(Object o){
    return $is.$1(o);
  }
}

interface $Cell_$Integer_$ {}

class $$Cell$_Integer_$ extends Cell
        implements $Cell_String_${
  static Cell.$IsolatedSnippets $is=null;
  $$Cell$_Integer_$(Object t){ super(t);}
  protected boolean $snip$1(Object o){
     return o instanceof Integer;
  }
}
```

```
class Pair extends Cell{
  private Object s;
  Pair(Object r,Object s){
     super(r); this.s=s;
  }
  void setSecond(Object s){ this=s; }
  Object getSecond(){ return s;}
  Pair getReverse(){ return $snip$2(s,get());}

  protected abstract boolean
            $snip$2(Object s,Object r);

  public abstract static class $IsolatedSnippets{
     boolean $2(Object o);
  }
}

interface $Pair$_Integer,String_$
                extends $Cell$_Integer${}

class $$Pair$_Integer,String_$ extends Pair
        implements $Pair$_Integer,String_${
  static Pair.$IsolatedSnippets $is=null;
  $$Pair$_Integer,String_$(Object r,Object s){
     super(r,s);
  }
  protected Pair $snip$2(Object s,Object r){
     return new $$Pair$_String,Integer_$(s,r);
  }
}

interface $Pair$_String,Integer_$
                extends $Cell$_String_${}

class $$Pair$_String,Integer_$ extends Pair
        implements $Pair$_String,Integer_${
  static Pair.$IsolatedSnippets $is=null;
  $$Pair$_String,Integer_$(Object r,Object s){
    super(r,s);
  }
  protected Pair $snip$2(Object s,Object r){
    return new $$Pair$_Integer,String_$(s,r);
  }
}
```

```
class Client {
  static {
    $$Cell$_String_$.$init(
        new Cell.$IsolatedSnippets(){
            boolean $1(Object o){
                return o instanceof String;
            }
        }
    );
  }
```

```
  public static void main(String[] s){
    Cell c1=new $$Cell$_Integer_$(null);
    c1.set(new Integer(1));
    Integer I=(Integer)c1.get();
    c1.setIfPossible("2");
    Cell c2=c1.getCString();
    Pair p=new $$Pair$_Integer,String_$(I,c2.get());
    Object o=p.getReverse();
    c1=o instanceof $Cell$_Integer_$? (Cell)o :null;
  }
}
```

Figure 3. *NextGen translation of classes Cell,Pair and Client*

```
\\Each parametric class will automatically implement this interface
interface $Parametric {
    public $Type getType();
}
\\The type descriptor class
class $Type {
    public Class type;  // The Class object of the generic class.
    public $Type[] ar;  // Type variables instantiation
    public $Type(Class t){ type=t; ar=new $Type[0];}
    public $Type(Class t,$Type[] a){ type=t; ar=a; }
    public boolean theSameOf($Type t){
        if (ar.length!=t.ar.length || !type.equals(t.type))  return false;
        for (int i=0;i<ar.length;i++)   if (!ar[i].theSameOf(t.ar[i])) return false;
        return true;
    }
    public boolean instanceOf(Object o){
        if (o instanceof $Parametric){
            $Parametric P=(($Parametric)o);
            $Type[] Par=P.getType().ar;
            if (Par.length>ar.length || !type.isInstance(o)) return false;
            for (int i=Par.length-1;i>=0;i--)
                if (!Par[i].theSameOf(ar[i])) return false;
        } else return type.isInstance(o);
    }
    public Object cast(Object o){
        if (instanceOf(o)) return o;
        else throw new ClassCastException();
    }
}
```

Figure 4. *Implementation of library classes for RM*

type descriptor `D` of class `$Type` is passed to the constructor, which stores it in a private field of `O`. `D` will be used both to perform instance tests and casts over `A<T>`, (see `$Type.istanceOf` method) and within `O` to access the instantiation of its type variables, via the public field `ar` of `D`. In Figure 5 the complete translation of the source code is shown. Notice how it makes use of the erasure technique, that is it *erases* the type variables to `Object` and a parametric class to its non-generic version (e.g. `Cell<Integer>` to `Cell`). The type descriptors management has only been inserted to support type operations.

It is not hard to see that this translation technique allows the full integration requested. However it has an important drawback: its run-time overhead is significant. First of all each time a type operation is required a new `$Type` object is created, possibly containing other `$Type` objects representing the instantiation of its type parameters, in a recursive way. Secondly an instance test is realized by comparing two descriptors. The overhead of both these tasks may be considered acceptable in the case of simple types such as `Cell<String>` or `Pair<Integer,String>`, but with complex structures of parameterization such as `Pair<Cell<String>,`
`Pair<Integer,Cell<String>>>` the influence on performance becomes much too relevant.
Our target then was to develop a translation based on the same mechanism, but which was able to bound run-time overhead at acceptable values, while not changing the general benefits described.

4 An approach based on load-time collection of type descriptors

Figure 6 characterizes current translations with respect to the overhead they introduce to support parameterized types. GJ does not carry any run-time information being realized by directly implementing the generic idiom, so it does not cause any overhead in both space (the class file size) and time (the execution speed). It has been represented with a white circle in order to highlight its lack of integration of parametric types with Java types. Full Heterogeneous Translation (HT) implementation of *Pizza*, allows such integration (black circle), but at the cost of a considerable space overhead, caused by the

```
class Cell extends $Parametric{
  private Object t;
  private $Type $c;
  Cell($Type $c,Object t){
    this.$c=$c;this.t=t;
  }
  void set(Object t){ this.t=t;}
  Object get(){ return t;}
  void setIfPossible(Object o){
    if ($c.ar[0].instanceOf(o))
          this.t=$c.ar[0].cast(o);
  }
  Cell getCString(){ return new Cell(
      new $Type{Cell.class, new $Type[]{
        new $Type(String.class)
      },""+t);
  }
  public $Type getType(){ return $c;}
}
```

```
class Pair extends Cell{
  private Object s;
  private $Type $c;
  Pair($Type $c,Object r,Object s){
    super(new $Type(Cell.class,
          new $Type[]{$c.ar[0]}
    ));
    this.$c=$c;this.s=s;
  }
  void setSecond(Object s){ this=s; }
  Object getSecond(){ return s;}
  Pair getReverse(){
    return new Pair(
      new $Type(Pair.class,
          new $Type[]{$c.ar[1],c.ar[0]},
      s,get());}
  public $Type getType(){ return $c;}
}
```

```
class Client{
    public static void main(String[] s){
        Cell c1=new Cell(new $Type(Cell.class,new $Type[]{new $Type{Integer.class}),null);
        c1.set(new Integer(1));
        Integer I=(Integer)c1.get();
        c1.setIfPossible("2");
        Cell c2=c1.getCString();
        Pair p=new Pair(new $Type(Pair.class,new $Type[]{
            new $Type{Integer.class},new $Type(String.class)}),I,c2.get());
        Object o=p.getReverse();
        c1=new $Type(Cell.class,new $Type[]{new $Type{Integer.class}}).instanceof(o)
                ?(Cell)new $Type(Cell.class,new $Type[]{new $Type{Integer.class}}).cast(o)
                :null;
    }
}
```

Figure 5. *RM translation of classes Cell,Pair and Client*

continuous creation of class files which are almost similar to each other, due to the use of different instantiations of a single generic class. Using a mixture of both approaches *NextGen* reduces the space overhead at the cost of a little overhead at run-time.
RM defers all the management of the instantiation of type variables at run-time, so it has a notable run-time overhead, but does not cause significant overhead in space[2].
The first idea one may think of applying to overcome the drawbacks of *RM*, is to create the type descriptors for the instantiations of the parametric classes one client refers to, at the client load-time. This can be realized by storing such `$Type` objects in static fields initialized by the static initializer of the client class[3]. For instance class `Client` may be translated as depicted in Figure 7.

This translation may prevent a descriptor from being created twice in a client, and moves the related overhead from run-time to load-time. This means that the code to create descriptors is not executed when needed, slowing down the run-time performance, but immediately after the loading of the client class which uses them. Considering that the time for a class file to be loaded (which involves file system access) is typically higher than the time to execute a little initialization code, such a management will decrease the influence on time performance.
The same technique may also be used to create the descriptor for `Cell<String>` within class `Cell`, but not for `Pair<S,R>` (method `getReverse`) and `Cell<S>` (constructor) in the class `Pair`. This is because the instantiation of their type variables depends on the current binding of the type variables of the class instantiation used. Furthermore such a technique is unable to make instance tests faster. We will call

[2] In fact RM increases a little the class files size due to the code added to manage type descriptors
[3] The Java run-time automatically calls the static initializer of a class after the class file is loaded

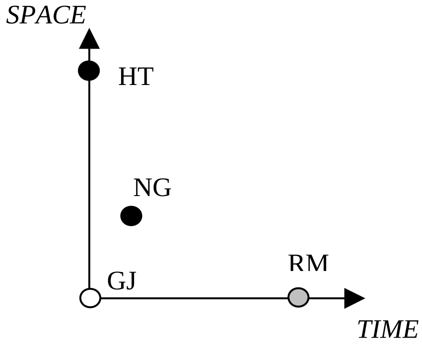

Figure 6. *Overhead caused by current translations*

```
class Client{
    private $Type[] $t=new $Type[2];
    static{
        $Type[][] $a=new $Type[][2];
        $a[0]=new $Type[]{new $Type{Integer.class};
        $a[1]=new $Type[]{new $Type{Integer.class},new $Type{String.class}};
        $t[0]=new $Type(Cell.class,$a[0]);
        $t[1]=new $Type(Pair.class,$a[1]);
    }
    public static void main(String[] s){
        Cell c1=new Cell($t[0],null);
        c1.set(new Integer(1));
        Integer I=(Integer)c1.get();
        c1.setIfPossible("2");
        Cell c2=c1.getCString();
        Pair p=new Pair($t[1],I,c2.get());
        Object o=p.getReverse();
        c1=$t[0].instanceof(o)?(Cell)$t.cast(o):null;
    }
}
```

Figure 7. *Optimized translation for class Client*

this intermediate translation *RLM* (Run-Time - Load-Time management translation).
In this section we will introduce a different translation, called *LM* (Load-Time management), which both permits the creation of all type descriptors at load-time, and allows instance tests to run faster. Considering the overhead as divided into the kinds: run-time, load-time and space, the situation will then be summarized as shown in Figure 8.

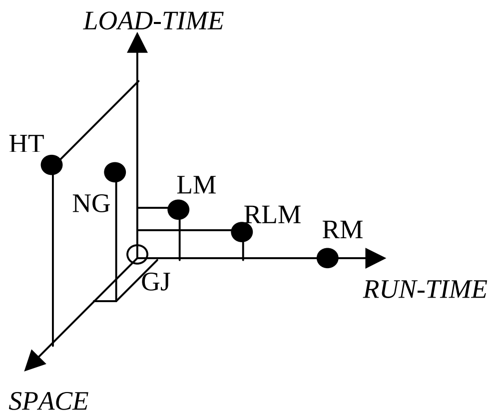

Figure 8. *Translations overhead with the new framework*

The idea of adding static initializers through translation to prepare useful information about the instantiation of parametric types is not brand new. In *NextGen* a similar technique was in fact used to manage the case in which a parametric type is instantiatied to a package-private class. Notice that such a problema does not hold for *LM*. In fact, the method `Class.isInstance` is the only functionality which we will apply to a type parameter outside the client class, and the Core Reflection is implemented so that it can be applied to the run-time representation of a package-private class, even within a different package.

4.1 Supporting type descriptors

One of our first goals is to prevent a given instantiation of a parametric type having two or more type descriptors living at run-time. For this reason we will use a library class called `$TDManager` (Type Descriptors Manager) that will accept registrations for type descriptors and keep track of them, preventing a descriptor from being created and stored twice. Methods and fields of `$TDManager` will be declared static. This allows the class generated by the translator to access the run-time data structure that `$TDManager` stores, by using the static methods it provides.
Descriptors type will be implemented with the library class `$TD` (Type Descriptor). As with `$Type` for *RM* translation, such a class will contain one field that stores the `Class` object of the type, and an array of `$TD` representing the instantiations of type variables.
To speed up the prevention of double registrations `$TDManager` will use a `Hashtable` structure containing `$TD` objects. When a client class wants to register a type descriptor, say the one for type `Cell<Integer>`, it will call a method `$TDManager.register` passing it the object `Integer.class` and an array of `$TD` which contains the type descriptor for class `Integer` (assuming it is already available). This method will check for the existence of a type descriptor with the same properties in the hash table, if it exists it is returned. Otherwise a new one is created, a unique identifier is assigned to it, it is stored in the hash table and finally it is returned.
In order to support this handling of registrations through the hash table, class `$TD` will provide methods `equals` and `hashcode` working over only two fields, the one representing the class, and the array of type descriptors[4].
Our second goal is to defer all the construction of type descriptors at load-time, even for those whose instantiation of type variables is not directly known using a static analysis, such as `Pair<S,R>` in method `Pair.getReverse`. In general a parametric class instantiation `T` will contain a set of types, that we call

[4]To this end only the identifier of type descriptors will be used

friend types of `T`, whose actual instantiation of type variables depends on the current binding of type variables of `T`. In our source code we have for example that `Cell<Integer>` and `Pair<String,Integer>` are friend types of `Pair<Integer,String>`.
When a client registers a parametric type, we need all its friend types to be automatically registered. This is achieved by delegating the registration of the instantiation of a parametric class `C` to a static method `createTD` of class `C`, which the translator automatically builds. By analyzing the source code of a given generic class the translator is able to insert the appropriate code in such a method, so that this task can be performed. The process of friend types creation will then proceed recursively. Notice that the direct super-type of a given parametric class is a special kind of friend type, that we call *father* type. It will be used to support the subtyping.

4.2 Implementation details

To summarize, *LM* translation is supported by three library classes (see Figure 9): `$TDManager`, `$TD` and `$Parametric`. `$TDManager` provides the run-time support to the translation. `$TD` is the class of all the type descriptors. It stores the `Class` representation of the current type, the type variables instantiation, the friend types descriptor and the father type descriptor. It provides methods to allow its correct handling by hash tables (`equals` and `hashcode`) and methods to resolve instance tests (`isInstance` and `navSubtype`). Field `bNew` together with method `checkNew` are used to aid the preventing of double registrations. Interface `$Parametric` will be automatically implemented by the generic classes, and allows fast retrieval of the type descriptor of an object created by a parametric type[5]. These three classes should be included into a package `java.generic` always accessible from the classes of the applications which use generic features. The `import` clause may be automatically inserted by the translator. Another library class which the translation will provide is `$Array`, in section 7.1 we will show how it is useful for handling arrays.
Application classes are changed by the translation in several ways as shown in the translation of the source code in Figure 10. A static initializer is added to each client class. It registers the type descriptors used by this class, and stores them in an array of static fields[6]. Simple types used as type variables are directly registered, while parametric types are registered using their `createTD` facility. Every time a parametric type is used in instantiations, casts or instance tests, such operations are translated into operations over the corresponding type descriptor, in the same way as discussed in Section 3 for *RM* translation.
Parametric classes store the type descriptor in the `$td` field, and permit it to be accessed using the `$Parametric.getTD` method. Within the constructor the first argument is stored in such field, and the father descriptor is used to call the super-constructor.
The static `createTD` method is added, and accommodates the creation of a descriptor. The first time a descriptor is registered the friend types and the father types are registered too. Finally all the uses of a type variable are translated into accesses to the corresponding descriptor.

Let us give an example to describe the dynamics of a typical pattern of creation and use of type descriptors.

- When the application starts, and the `$TDManager` is accessed by some client class, it is loaded and initialized

- When class `Client` is loaded its static initializer is executed, and this causes some type to be registered. When `Pair<Integer,String>` is registered, method `Pair.createTD` is called by passing it the two descriptors for `Integer` and `String`, that at this time are already available.

- Method `Pair.createTD` registers a newly created descriptor, if it has already been created the methods suddenly returns.

[5]We could avoid the use of such interface and retrieve the type descriptor using the reflective feature which accesses the fields of an object, but this would be too much expensive

[6]We used a field of type array instead of different fields just for compactness. As a matter of fact the solution which uses different fields leads to better run-time performance

```
public class $TD {
  public int ID;  //Unique identifier
  public Class c;
  public $TD[] params,friends;
  public $TD father;
  private int hashValue;
  private boolean bNew;
  public static $TD[] voidTDs=new $TD[0];
  public static $TD objTD;

  static {  //Initializes objTD voidTDs
     objTD=new $TD(Object.class,voidTDs);
     objTD.ID=0;  objTD.bNew=true;
     objTD.father=objTD;
  }
  public $TD(Class c, $TD[] p) {
     this.c=c;  params=p;   bNew=true;
     friends=null;  father=null;  ID=-1;
     processHash();
  }
  public boolean checkNew() {
     if (!bNew) return bNew;
     bNew=false;  return true;
  }
  private void processHash() {
     hashValue=c.hashCode();
     for (int i=params.length-1;i>=0;i--)
         hashValue+=params[i].ID;
  }
  public int hashCode() { return hashValue; }

  public boolean equals(Object obj) {
     if (!(obj instanceof $TD))
        return false;
     $TD S=(($TD)obj);
     if (!S.c.equals(c)) return false;
     if (S.params.length!=params.length)
         return false;
     for (int i=params.length-1;i>=0;i--)
         if (S.params[i].ID!=params[i].ID)
             return false;
     return true;
  }
  // Implementing subtyping
  public boolean isInstance(Object o) {
     return o instanceof $Parametric?
       navSubtype((($Parametric)o).getTD(),ID)
       :c.isInstance(o);
  }
  public static boolean
        navSubtype($TD P,int pos) {
     int Q;
     while ((Q=P.ID)!=pos && Q!=0) P=P.father;
     return Q==pos;
  }
}
```

```
public class $TDManager {
  private static Hashtable h;
  public static int countID;
  static{
     h=new Hashtable(101,0.7f);
     h.put($TD.objTD,$TD.objTD);
     countID=1;
  }
  public static $TD register(Class c) {
     $TD T=new $TD(c,$TD.voidTDs);
     Object o=h.get(T);
     if (o==null){
         h.put(T,T);
         T.ID=countID++;
         T.father=register(c.getSuperclass());
         return T;
      }
      return(($TD)o);
  }
  public static $TD register(Class c,$TD[] p) {
      $TD T=new $TD(c,p);
      Object o=h.get(T);
      if (o==null){
          h.put(T,T);
          T.ID=countID++;
          return T;
      }
      return(($TD)o);
  }
}

public interface $Parametric {
    public $TD getTD();
}

public class $Array implements $Parametric {
  public static $TD createTD($TD[] params) {
    $TD t=$TDManager.register(
        $Array.class,params);
    if (t.checkNew()){
      t.father=$TD.objTD;
      t.friends=$TD.voidTDs;
    }
    return t;
  }
  public $TD getTD() { return $td;}
  private $TD $td;
  private Object[] ao;
  $Array($TD $td,int n){
    this.$td=$td; ao=new Object[n];
  }
  $Array($TD $td,Object[] a){
    this.$td=$td; ao=a;
  }
  void set(Object o,int n){ ao[n]=o;}
  Object get(int n){ return ao[n];}
}
```

Figure 9. *Implementation for library classes*

```
public class Cell implements $Parametric {
  static $TD $cl;
  static{
    $cl=createTD(new $TD[]{
      $TDManager.register(String.class)});
  }
  public static $TD createTD($TD[] params) {
    $TD t=$TDManager.register(
                Cell.class,params);
    if (t.checkNew()){
      t.father=$TD.objTD;
      t.friends=$TD.voidTDs;
    }
    return t;
  }
  public $TD getTD() { return $td;}
  private $TD $td;

  private Object t;
  Cell($TD $td,Object t){
    this.$td=$td; this.t=t;
  }
  void set(Object t){ this.t=t;}
  Object get(){ return t;}
  void setIfPossible(Object o){
    if ($td.params[0].isInstance(o)) t=o;
  }
  Cell getCString(){
    return new Cell($cl,""+t);
  }
}
```

```
public class Pair extends Cell {

  public static $TD createTD($TD[] params) {
    $TD t=$TDManager.register(Pair.class,params);
    if (t.checkNew()){
      t.father=Cell.createTD(new $TD[]{params[0]});
      t.friends=new $TD[1];
      t.friends[0]=Pair.createTD(
        new $TD[]{params[1],params[0]});
    }
    return t;
  }
  public $TD getTD() {return $td;}
  private $TD $td;

  Object s;
  public Pair($TD $td,Object r,Object s) {
    super($td.father,r);
    this.$td=$td; this.s=s;
  }
  public void setSecond(Object s) { this.s=s; }
  public Object getSecond() { return s;}
  public Pair getReverse() {
    return new Pair($td.friends[0],s,get());
  }
}
```

```
public class Client {

    static $TD[] $t=new $TD[4];
    static{
        $t[0]=$TDManager.register(Integer.class);   //Integer
        $t[1]=Cell.createTD(new $TD[]{$t[0]});      //Cell<Integer>
        $t[2]=$TDManager.register(String.class);    //String
        $t[3]=Pair.createTD(new $TD[]{$t[0],$t[2]});//Pair<Integer,String>
    }

    public static void main(java.lang.String[] args) {
        Cell c1=new Cell($t[1],null);        // new Cell<Integer>(null);
        c1.set(new Integer(1));
        Integer I=(Integer)c1.get();
        c1.setIfPossible("2");
        Cell c2=c1.getCString();
        Pair p=new Pair($t[3],I,c2.get());   //new Pair<Int,Str>(I,c2,get());
        Object o=p.getReverse();
        c1=$t[1].isInstance(o)?(Cell)o:null; // o instanceof Cell<String>?...
    }
}
```

Figure 10. *Translation of the source code*

- Otherwise the $TDManager insert it into the hash table and assign it a new identifier. createTD creates then the descriptor for Cell<Integer> (as we were in a client class for it) and stores it into the father field. Then it creates the descriptor for Pair<String,Integer> and stores it into the friends field.

- When registering Pair<String,Integer> the method Pir.createTD tries to register the descriptor for Pair<Integer,String> again, but this time the $TDManager prevents it from being created twice.

- After some more registration for type descriptor the load of Client finishes, and $TDManager will store all the descriptors that Client needs.

- Within method Client.main, when creating the object that will be assigned to the variable p, the descriptor for Pair<Integer,String> is passed to the constructor.

- In the method p.getReverse the descriptor for Pair<String,Integer> is accessed through the friend attribute and is used to create the reversed pair.

4.3 Handling Subtyping

We already argued that the integration between parametric types and Java typing with the framework of homogeneous translation could be achieved by realizing an instanceof primitive, which by accepting an object O and the descriptor of a type T(a generic type, a simple type, or a type variable) checks if O belongs to T. With the translation we are describing, this functionality is stored in the method $TD.isInstance. If the object does not implement the interface $Parametric it means that none of its parents is a generic class, so the instance test can be realized by directly using the reflective functionality Class.isInstance. In the opposite case it is possible to get its $TD representation with the method $Parametric.getTD after a cast (narrowing), and use it to check if the object type To is a subtype of the receiver type of the instance test, T[7]. In the code given above this is realized by navigating the inheritance chain from To to Object using the $TD.father attribute and searching for T[8].
Such a technique has the main drawback of an execution time of instance test which depends on the length of the inheritance chain between the type Object and the receiver type.
Another implementation may work by storing subtyping information into each type descriptor during its creation, so that instance tests may be realized with a direct access. For example each $TD object could keep a sequence of bits, which we call SBR (Subtyping Binary Representation), in which a bit set at position i means that the $TD object with ID=$i$ is a supertype of the current one. Such a data structure can be implemented as a byte[] field of type $TD , it has to be built in the createTD method of each parametric class, and in the register method of class $TDManager to handle non-generic types. Instead of using the navSubtype method of class $TD we will use a directSubtype method which only accesses the right bit. A straightforward implementation may be the one of Figure 11. Method getObjectSBR is used to create the subtyping info for the *Object* descriptor. When a new SBR array has to be created (within method createTD for parametric type and within method $TDManager.register for simple types) the copySBR method is applied to the SBR field of the father descriptor, and then setSBRbit is applied to add the bit relative to the current descriptor. Notice that copySBR does not only duplicate the array, but also adds some zero in order to deal with the increment of the *ID* counter of the $TDManager .

5 Performance and Optimization

One of the principal issues that constrained the development of this translation and in general of all the extensions realized with translation approaches is the performance. As already mentioned performance

[7] In the operation "o instanceof T" we will call o the receiver object of the test, and T the receiver type of the test

[8] For clarity Ii the code we used the method $TD.navSubtype, but actual implementation should use only a unique final static method to improve performance

```
class $TD{
    ...
    public byte[] SBR; //Subtyping Binary Representation

    final public static byte[] getObjectSBR(){
        return new byte[]{(byte)1}; // [00000001]
    }
    final public static byte[] copySBR(byte[] b){
        SBR=new byte[(($TDManager.countID-1)>>3)+1];
        for (int i=SBR.length-1;i>=0;i--)
           SBR[i]=(i>=b.length?0:$b[i]);
        return SBR;
    }
    final public static boolean getSBRbit(byte[] b,int pos){
        return ( (pos>>3)<b.length && (b[pos>>3] & (1<<(pos & 7)))!=0);
    }
    final public static void setSBRbit(byte[] b,int pos) {
        b[pos>>3] |= (1 << (pos & 7));}
    }
    public static final boolean directSubtype($TD P,int pos) {
        return getSBRbit(P.SBR,pos);
    }
}
```

Figure 11. *Implemantation details of direct subtyping management*

was one of the main reasons why *RM* translation was never proposed, even if it could solve both the main drawbacks of the other approaches, namely the lack of type integration of *GJ* and the code expansion of *NextGen*. With the *LM* translation we obtained all the advantages of *RM* in a framework in which application performances are not affected in a significant manner.
In fact our special management of type descriptors allows fast implementation of type-operations, and little impact on the load-time. Our original target was to develop a translation whose space and time overhead was not greater than the 10% of the performance of the code written using the generic idiom, that is the *GJ* technique[9].
With respect to performance the run-time environment may be characterized by the following four aspects: run-time execution (the time to run the application), load-time (the time to load the applications class files), the memory usage of the application and finally the storage dimension of the application (class files size). Our target was not to overcome the overhead limit (10%) in any of these four aspects. Some of these kinds of overhead are not fixed, but depends on the specific applications, and particularly on the uses they make of parametric types. For this reason we will sometime refer to some applications and classes that we used as benchmarks.

Application size

The class files increased by the translation are:

- Client classes: which include static attributes to access the type descriptors used, a static initializer which initializes them, and a translation of type operations which is in general augmented with a little code. We used classes which referred from 3 to 15 parametric types, each having from 1 up to 3 type variables, obtaining a class file increment from 1 kbyte up to 4 kbytes.
- Parametric classes: which are augmented with the *$td* attribute, with its getter method and its setting code. They also have the facility method *createTD*, to handle the registration of types. We obtained, depending on the number of type parameters, from 1 up to 3, and on the number of friend types, form 0 to 5, an overhead from 1 kbyte up to 2kbytes.

[9]Notice that this was the original requirement Java designers asked for in order to accept contribution to extending Java with genericity

Memory usage

Comparing the memory used by the application with respect to *GJ* framework, the main overhead is the one caused by the `$TDManager` data structure, which keeps track of all the `$TD` objects. Such overhead directly depends on the number of descriptors the applications use, and on their complexity (mainly the number of friend descriptors they have). Another cause of memory overhead is the field `$td`, which each instance of a parametric class carries. We took measures from applications of different sizes:

- Small application: composed by 1 class which causes 10 descriptors to be registered (each with up to 5 friend types), obtaining a memory overhead of 500 bytes.
- Medium application: composed by 5 classes which register 50 descriptors, causing a memory overhead of 4 kbytes.
- Medium-Large application: composed by 15 classes which register 150 descriptors, with a memory overhead of 15 kbytes.

Load-Time In a typical Java framework when a class is loaded its class file is read from the storage (local or remote mass storage), a run-time representation of it is created, and finally its static initializer and all the initializations of static fields are executed. Our translation influences in a significant way the third of these steps for client classes only. In their static initializer a set of descriptors are registered, and the `$TD` objects obtained are stored in static fields.
The time to execute the initialization code has been measured on the three classes previously described for the Application Size overhead, obtaining a time execution from $10\mu secs$ to $200\mu secs$. The time to load the class, which depends on the kind of storage (we used a common local hard disk) goes from $5msecs$ to $30msecs$[10]. Notice that *NextGen* has a much higher load-time cost, due to the need to load one class and one interface for each parametric type instantiation, involving new storage access.

Run-Time The two operations whose translation may affect performances are parametric type object creation and instance tests[11].

- Instantiation: In general the coast of a class instantiation depends on its number of fields and super-classes and on the complexity of the constructor code. Our translation adds to each parametric class the field `$td`, a new interface implemented called `$Parametric`, and in the constructor one more argument and the code to put it in the new field. In the case of a trivial generic class without attributes, methods, and constructor code, the time overhead has been revealed to be about the 15%. In real cases just by adding a couple of fields and the code to initialize them the overhead becomes less than the 5%.
- Instance test: Out main concern was to develop a translation which could implement instance tests in a fast way. The principal problem was that Java realizes such tests at very low cost, which is comparable to that of method calls, or field accesses. If the receiver object of the instance test is not an instantiation of a parametric class then reflective method `Class.isInstance` is called. If the object implements the `$Parametric` interface instead, we may apply the two different methodology discussed: navigational or direct. The results are depicted in Figure 12. The values are normalized to the cost of the core instance test[12]. With

JVM Architecture	core *instanceof*	*Class. isInstance*	Navigation Normal	Navigation Optimized	Direct Normal	Direct Optimized
JDK1.1.7	1.0	2.0	11.2	4.8	6.4	4.8
JDK1.2 (JIT)	1.0	75.0	70.0	23.0	93.0	25.0

Figure 12. *Performance measurement of the translation of instance tests*

[10] Measures performed over a PentiumII 350MHz with Windows NT O.S.
[11] Casts operations are directly realized using instance tests, so their impact is analogous.

the label "optimized" we considered the translation that inlines all the code needed to realize the test.

For the application size, the memory usage and the load-time, we obtained good performance results. Typical applications will have an overhead lower than the 10% limit we set, and only in degenerating cases will such a limit be overcome. Some more words have to be spent on the run-time aspect, and in particular for the instance test operation.
It seem to be the only point in which the overhead does not satisfy. Notice however, that the instance test over parametric types and type variables, is an operation that is not provided by `GJ`, and that `NextGen` is able to implement them with no overhead at run-time through the heterogeneous approach, leading to a load-time overhead. Anyway our experience with the use of this framework has shown that the full integration of parametric types with the core language, which is useful for expressiveness reasons, will not cause significant performance overhead in the overall execution.
Let us now consider the differences found using the two most common Virtual Machines, that of jdk (Java Development Kit) version 1.1 ([JDK1.1]) which has a standard compiler and version 1.2([JDK1.2]) which has a JIT(Just In Time) compiler. Application size, memory overhead, and run-time cost of class instantiation are more or less analogous. Load-Time overhead results as being lower with jdk1.2, since while the load of the class becomes slower due the compilation being executed "In-Time", the execution of static initializers becomes faster due to native execution.
For instance tests, in the JIT compiler of jdk1.2 the cost of the `Class.isInstance` method remains the real bottleneck, our translation being faster when dealing with parametric types. As one may notice the instanceof core test is highly optimized by the JIT architecture, in which it is up to 50 times faster with respect to the version of the jdk1.1.7, while the cost of the method `Class.isInstance` is almost unchanged. Using the jdk1.1.7 compiler instead, the difference between the cost of the translated version of the instance test and the cost of the `Class.isInstance` method is higher, but the former is more similar to that of the core instance test.

6 On the recursive creation of parametric types

We saw that our translation creates at load-time one descriptor object for each instantiation of parametric type which is used by the application in a type operation; that is in generic object creations, in instance tests and in casts. *HT* approaches, and *NextGen*, behave in a similar way. For each instantiation of a parametric type which is used somewhere in a program, they create one class file at load-time. So, one important issue that must be considered is whether the set of parametric types a program can refer, directly or not, is finite.
It is not hard to show that in general such a set is not finite, as in the case in which class `Cell<T>` has the code to create an object of type `Cell<Cell<T>>`. In this case the type `Cell<String>` for instance would refer the type `Cell<Cell<String>>` which refers the type `Cell<Cell<Cell<String>>>`, and so on. In this situation the *LM* translation would try to build at load-time an infinite hash table of `$TD` objects, while heterogeneous translations (and *NextGen*) would try to create an infinite set of class files. Approaches that do not carry any information about the instantiation of type parameters such as *GJ*, or which creates it when needed such as both *RLM* and *LM* do not suffer from this problem.
So we are interested in the existence of a property which can be useful to determine if a given program contains an infinite set of referenced types, i.e. types used by some client class. Such a property will be used to detect early translation termination failings.
As far as we know previous papers about parametric polymorphism in Java did not mention this issue, which leads to a difference of expressiveness between the various implementations of parametric polymorphism. Moreover some widely used implementation of parametric types in other programming languages, like *C++*, does not apply any control to intercept this problem, causing the compilation of some program simply failing to terminate.

[12] These costs are 250nsec in the jdk1.1.7 and 4nsec in the jdk1.2, using the architecture already mentioned

6.1 The formalization of the problem

A finite set of parametric classes will be called *closed* if none of them refers the instantiation of a parametric class out of the set. This property grants that such a set can be analyzed and compiled without any information about other classes.
We will suppose that such a program could refer only a finite set of simple types $\mathcal{S}$ (`Integer`,`String`,..).
Let $\mathcal{G}$ be the set of names of the closed set of parametric classes we are interested in (`Cell`,`Pair`,..),$\forall g \in \mathcal{G}$ we denote with n_g its number of parameters.
We call a *full instantiation* in $\mathcal{G}$ any element of the set $\mathcal{I}^{\mathcal{G}} = \{(g,d) : g \in \mathcal{G}, d : \{1..n_g\} \mapsto \mathcal{S} \cup \mathcal{I}^{\mathcal{G}}\}$. Any couple $(g,d) \in \mathcal{I}^{\mathcal{G}}$ will be called a full instantiation of g. For example a full instantiation of `Pair` may be `Pair<String,Cell<Integer>>`; here $g =$ `Pair` and d associates to the first parameter `String`($\in \mathcal{S}$) and to the second `Cell<Integer>`($\in \mathcal{I}^{\mathcal{G}}$).
We call an *instantiation dependent* from n variables any element of the set $\mathcal{I}^{\mathcal{G}}_n = \{(g,d) : g \in \mathcal{G}, d : \{1..n_g\} \mapsto \mathcal{S} \cup \{1..n\} \cup \mathcal{I}^{\mathcal{G}}_n\}$. In this case function d can associate to each parameter a simple type ($\mathcal{S}$) or one of the n free type variables ($\{1..n\}$ 1 for the first, .., n for the nth) or an instantiation in $\mathcal{I}^{\mathcal{G}}_n$ again.
`Pair<`T_1`,Cell<`T_2`>>` may be considered an instantiation dependent from two variables, T_1 and T_2.
To check if such an instantiation effectively uses one of the type variables we will use the following predicate:

$$\forall (g,d) \in \mathcal{I}^{\mathcal{G}}_n \ \ \mathcal{H}as((g,d),k) \Leftrightarrow k \in \mathcal{R}ng(d) \vee \exists j \ (d(j) \in \mathcal{I}^{\mathcal{G}}_n \wedge \mathcal{H}as(d(j),k)).$$

Notice that $\mathcal{I}^{\mathcal{G}} = \mathcal{I}^{\mathcal{G}}_0 \subset \mathcal{I}^{\mathcal{G}}_i \subset \mathcal{I}^{\mathcal{G}}_{i+1} \forall i$.
An element of $\mathcal{I}^{\mathcal{G}}_n$ can be *fully-instantiated* using the following substitution function $\mathcal{S}ub$:

$$\mathcal{S}ub : \mathcal{I}^{\mathcal{G}}_n \times \{\mathcal{I}^{\mathcal{G}} \cup \mathcal{S}\}^n \mapsto \mathcal{I}^{\mathcal{G}}$$

$$\mathcal{S}ub((g,d),i_1,..,i_n) = (g, \begin{cases} k \mapsto d(k) & \text{if } d(k) \in \mathcal{S} \\ k \mapsto i_{d(k)} & \text{if } d(k) \in \{1..n\} \\ k \mapsto \mathcal{S}ub(d(k),i_1,..,i_n) & \text{if } d(k) \in \mathcal{I}^{\mathcal{G}}_n \end{cases})$$

For example, if $i \in \mathcal{I}^{\mathcal{G}}_1$ represents the (non full) instantiation `Pair<Integer,Cell<`T_1`>>`, and $par \in \mathcal{I}^{\mathcal{G}}$ represents `Cell<String>` (which is the instantiation we want to apply to T_1), then $\mathcal{S}ub(i, par) \in \mathcal{I}^{\mathcal{G}}$ will represent the full instantiation `Pair<Integer,Cell<Cell<String>>>`.
In previous chapters we saw that a parametric class can refer in its code a set of other parametric classes, whose instantiation can possibly depend from the current type variables binding. This idea can be mapped by denoting the set of the type *directly-referenced* by g with a finite set $r_g \subset \mathcal{I}^{\mathcal{G}}_{n_g}$. For example $r_{Cell<T>}$ may be {`Cell<String>`, `Pair<T,T>`, `Pair<T,String>`, ..}. Given an instantiated type $i \in \mathcal{I}^{\mathcal{G}}$ we use function $\mathcal{D}ir$ to retrieve the full instantiation it directly refers:

$$\mathcal{D}ir : \mathcal{I}^{\mathcal{G}} \mapsto \mathcal{P}(\mathcal{I}^{\mathcal{G}})$$

$$\mathcal{D}ir((g,d)) = \{\mathcal{S}ub(i, d(1), .., d(n_g) : i \in r_g\}$$

With all these definitions is now possible to define the set of *dependent* (or referenced) types of a given full-instantiation in $\mathcal{G}$, that is a sort of transitive closure of the "direct-reference" relation, as:

$$\mathcal{D}ep : \mathcal{I}^{\mathcal{G}} \mapsto \mathcal{P}(\mathcal{I}^{\mathcal{G}})$$

$$\mathcal{D}ep((g,d)) = (g,d) \cup \bigsqcup_{i \in r_g} \mathcal{D}ep(\mathcal{S}ub(i, d(1), .., d(n_g)) = (g,d) \cup \bigsqcup_{i \in \mathcal{D}ir((g,d))} \mathcal{D}ep(i)$$

Given a program which contains parametric classes $\mathcal{G}$ its translation (in *LM* and *NextGen*) do not fails to terminate if $\forall i \in \mathcal{I}^{\mathcal{G}}, \mathcal{D}ep(i)$ is a finite set. In such cause we say that it is *generating-safe*.
In order to introduce a condition for a program to be generating-safe we need to build a graph structure

```
class A<T>{
\\ has the referenced type B<T,T>
... } class B<R,S>{
\\ has the referenced types A<S>,
\\ B<S,String>, C<R,Integer>
... } class C<R,S>{ ...
\\ has the referenced types
\\ C<String,A<R>>,B<String,S>
... }
```

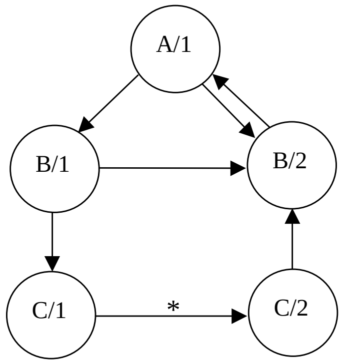

Figure 13. *Type variable dependencies - An example*

which represents the dependencies between parametric types and their type variables. This graph is a triplet $(\mathcal{H}, E, E^*)$ with $\mathcal{H} = \{(g,n) : g \in \mathcal{G}, 1 \leq n \leq n_g\}$ and $E, E^* \subset \mathcal{H}x\mathcal{H}$, such that:

$$((h,n),(l,m)) \in E \Leftrightarrow \exists (l,d) \in r_h \ \ d(m) = n$$

$$((h,n),(l,m)) \in E^* \Leftrightarrow \exists (l,d) \in r_h \ \ \mathcal{H}as(d(m),n) \wedge d(m) \neq n$$

Such a graph has generic class type parameters as nodes and two kinds of oriented arcs, that we call normal arcs and *-labelled arcs. A normal arc goes from the i^{th} type variable of a class C to the j^{th} type variable of `D` if `C<`T_1`,..,`T_i`,..,`T_n`>` has a referenced type of the kind `D<`R_1`,..,`T_i`,..,`R_m`>` with T_i in j^{th} position. A *-labelled arc goes from the i^{th} type variable of a class `C` to the j^{th} type variable of `D` if `C<`T_1`,..,`T_i`,..,`T_n`>` has a referenced type of the kind `D<`R_1`,..,X<..,`T_i`,..>,..,`R_m`>` with `X` a parametric class and `X<..,`T_i`,..>` in j^{th} position of `D`. We call such a graph the type variable dependencies graph associated to the program.
In Figure 13 is shown an example of type variable dependencies and the relative graph.

6.2 Early detection of the problem

Giving an oriented graph (F, H) with F set of nodes and $H \subset FxF$ set of edges, we will denote with $\mathcal{C}l(F, A)$ the set of the cycles of the graph.

$$\mathcal{C}l(F,A) = \{(f_1, .., f_k) \in F^k, k > 1 : \forall i \neq j \ \ f_i \neq f_j \wedge \forall i \ \ (f_i, f_{i+1}) \in A \wedge (f_k, f_1) \in A\}$$

Theorem 6.2.1 *A closed set of parametric classes $\mathcal{G}$ is generating-safe iff its graph structure has not cycles with a *-labelled cycle (a cycle with at least one *-labelled arc), i.e.:*

$$\exists i \in \mathcal{I}^{\mathcal{G}}, \mathcal{D}ep(i) \ \textit{is not finite} \Leftrightarrow \mathcal{C}l(H,E) = \mathcal{C}l(H, E \cup E^*) = \mathcal{C}l(H, E \setminus E^*)$$

PROOF. The right part of the relation asserts that adding a *-labelled arcs would not produce new cycles, and drop them would not drop cycles. This is equivalent to say that none cycles has a *-labelled arc.
First, we proof the sufficiency. Suppose $((g_1,n_1), .., (g_k,n_k))$ makes up a cycle on the $(H, E \cup E^*)$ structure, and that $((g_j,n_j),(g_{j+1},n_{j+1})) \in E^*$ with $j < k$. Starting from a generic full-instantiation of g_1 in which $T_1, .., T_{n_{g_1}}$ are elements in $\mathcal{S} \cup \mathcal{I}^{\mathcal{G}}$ we can write a sequence in $\mathcal{I}^{\mathcal{G}}$ in the following way:

$$i_1 = g_1 < T_1, .., T_j >$$
$$i_2 = g_2 < X_2^1, .., X_2^{n_2-1}, T_{n_1}, X_2^{n_2+1}, .., X_2^{n_{g_2}} >$$
$$i_3 = g_3 < X_3^1, .., X_3^{n_3-1}, T_{n_1}, X_3^{n_3+1}, .., X_3^{n_{g_3}} >$$
$$..$$
$$i_j = g_j < X_j^1, .., X_j^{n_j-1}, T_{n_1}, X_j^{n_j+1}, .., X_j^{n_{g_j}} >$$
$$i_{j+1} = g_{j+1} < X_{j+1}^1, .., X_{j+1}^{n_{j+1}-1}, x < .., T_{n_1}, .. >, X_{j+1}^{n_{j+1}+1}, .., X_{j+1}^{n_{g_{j+1}}} >$$
$$..$$
$$i_k = g_k < X_k^1, .., X_k^{n_k-1}, x < .., T_{n_1}, .. >, X_k^{n_k+1}, .., X_k^{n_{g_k}} >$$
$$i_{k+1} = g_1 < X_{k+1}^1, .., X_{2k+1}^{n_1-1}, x < .., T_{n_1}, .. >, X_{k+1}^{n_1+1}, .., X_{k+1}^{n_{g_1}} >$$
$$..$$
$$i_{2k+1} = g_1 < X_{2k+1}^1, .., X_{2k+1}^{n_1-1}, x < .., x < .., T_{n_1}, .. >, .. >, X_{2k+1}^{n_1+1}, .., X_{2k+1}^{n_{g_1}} >$$
$$..$$

Because of the way it has been built this infinite sequence of elements of $\mathcal{I}^{\mathcal{G}}$ is such that for opportune $X_l^m \in \mathcal{I}^{\mathcal{G}}$ and $x \in \mathcal{G}$ all its elements are differents and $\forall l, i_l \in \mathcal{D}ir(i_{l-1})$. So we have also $\forall\, l, i_l \in \mathcal{D}ep(i_1)$ and hence $\mathcal{D}ep(i_1)$ is not finite. Things do not change if a cycle contains more than one *-labelled arc. For example in the program of Figure 13 we have the cycle $A/1, B/1, C/1, C/2^*, B/2, A/1$ and we can write the sequence:

```
A<T>,B<T,T>,C<T,Integer>,C<String,A<T>>,B<String,A<T>>,A<A<T>>
```

and recursively reach `A<A<T>>`, `A<A<A<T>>>` and so on.
So to speak, the type parameter T flows through the sequence following the dependencies graph path, being overheaded with other stuff (in this case with an `A<.>`), each time it transits across a *-labelled arc of the graph.

Now we have to proof that the condition is also necessary. Suppose that the graph has not cycles with *-labelled arcs. Consider we want to build a sequence $(i_k = (g_k, d_k) \in \mathcal{I}^{\mathcal{G}} : k > 0)$ like the preceding, so that $(i_k) \in \mathcal{D}ir(i_{k-1})$. We want to show that such a sequence cannot contain an infinite set of different elements, i.e. $\{i_k : k > 0\}$ is finite.
Transiting from one element to the successive into this sequence means to make each type variable transits to a new node of H through an arc, or to make a type variable be discarded. Hence we can characterize the sequence of transitions with the following sequence:

$$(tr_k : \{1..n_{g_k}\} \mapsto E \cup E^* \cup \{\oslash\}, k > 0)$$

which assigns to the type variables instantiation of each i_k an arc, a *-labelled arc or a $\oslash$ element, which represents the discarding. Notice that if the graph has no cycles with *-labelled arcs then each type variable will transit a finite number of times through such *-labelled arcs.
Now consider the following four sets:

- The starting type variables, that is the type variables instantiation of the first element of the sequence: $R_1 = \mathcal{R}ng(d_1)$
- The set of simple types: $R_2 = \mathcal{S}$
- All the elements of $\mathcal{I}^{\mathcal{G}}$ that occurs as type variables instantiation into the referenced types r_g of all $g \in \mathcal{G}$, that is: $R_3 = \bigcup_{g \in \mathcal{G}} \bigcup_{(h,d) \in r_g} \mathcal{R}ng(d) \cap \mathcal{I}^{\mathcal{G}}$
- All the types that will be created due to the transit of one or more parameters across *-labelled arcs: $R_4 = \bigcup_{k>1} \{d_k(i) : \exists h \in H \quad (h, (g_k, i)) \in \mathcal{R}ng(tr_{k-1}) \cap E^*\}$

The set of values that the type parameters of all the elements of the sequence can be instantiated to, I_t, are certainly a subset of the union of these sets, that is:

$$I_t = \bigcup_{k>0} \{d_k(i)\} \subset R_1 \cup R_2 \cup R_3 \cup R_4$$

All these four sets are finite, so I_t is finite. The following relation holds:

$$\{i_k : k > 0\} \subset \{(f, c) : f \in \mathcal{G} \wedge c \in \{1..n_f\} \mapsto I_t\}$$

The number of elements of the sequence is not greater than the number of all the elements of $\mathcal{I}^{\mathcal{G}}$ obtained by instantiating each $g \in \mathcal{G}$ with all the elements of the four sets above.
Now consider a tree which has i^o as root, each node n has all of its directly referenced types $\mathcal{D}ir(n)$ as sons, omitting those that are parents of n. Each path from the root to a leaf correspond to a sequence as we described above, it contains only elements of the sequence and these are all distinct to each other. Having such a sequence a finite number of elements, the path is finite and the tree has a finite depth. The set of nodes of the tree, that is nothing but $Dep(i^o)$, is finite. Such property holds for each $i^o \in \mathcal{I}^{\mathcal{G}}$.
□

7 Implementing related language features

In previous sections we sketched the general behaviour of our translator. We show how it handles type dependent operations over parametric types and type variables, and how it deals with inheritance. In this section we intend to describe how our translator can be accommodated in order to support other features related to the parametric polymorphism. For space reasons our description will be necessarily concise.

7.1 Arrays

Arrays are a case of parametric types already supported by the language; the type of the elements contained in the array is the type from which the array abstracts. In particular they are managed in the same way our translator manages new parametric types, that is by carrying the instantiation of type variables at run-time, allowing instance tests and casts.
The direct integration of arrays in our framework can be implemented by providing a new kind of class `$Array` managed as *LM* does with parametric types, which will be used as a wrapper for core arrays. In Figure 9 we gave the implementation of such a class which will be used for this target. It has a field of type `Object[]` which will contain the core array, and has getter and setter methods to access the elements of the array. It also provides two constructors, one which is the mapping of the core array creation (for example in `new Integer[3]`), and the other to create an `$Array` object given a core array. Being managed as a parametric type, class `$Array` will provide the field `$td`, the method `createTD`, and its constructors will be implemented using the usual translation pattern. Notice that the `$Array` implementation is exactly the translation of a parametric wrapper for arrays one would write. By managing arrays using such a mapping technique, the translation can obtain a perfect integration, being able to use parameterization structures such as `Cell<String[]>` or `Cell<String>[]`, that will be translated as if one had used respectively `Cell<$Array<String>>` and `$Array<Cell<String>>`. In Figure 14 a source code and a translation for a client class that uses arrays, is given. Notice that if one tries to get or set an element exceeding the array bound, they obtain the usual out-of-bound exception[13] as if they were using core arrays. The constructor of class `$Array` which takes a core array, is used in the cases in which an existing core array should be used when an `$Array` is expected.
The overhead caused by this kind of management, is analogous to that introduced by mapping primitive types into wrapper classes as `GJ` and `NextGen` do, which is required by the erasure technique.
Java arrays are treated with co-variance, that is the type system assumes `A[]` is a subtype of `B[]` when `A` inherits from `B`. As the authors of previous proposals did, we won't provide co-variance for arrays, since we believe that the complication needed for the related implementation is not worth the advantages one obtains. However we think that the problem of the application of co-variant and contra-variant subtyping to parametric types (see [Mey86] and [Coo89]) deserves further study.

[13] Which in Java is the java.lang.ArrayIndexOutOfBoundException

```
class ArrayClient{
  public static void main(String[] s){
    Cell<String>[] ac=new Cell<String>[1];
    ac[1]=new Cell<String>("1");
    String s=ac[1].get();
    String[] s=new String[0];
    Cell<String[]> cs=new Cell<String[]>(s);
  }
}
```

```
class ArrayClient{
  public static void main(String[] s){
    $Array ac=new $Array($t[4],1);
    ac.set("1",1);
    String s=(String)((Cell)ac.get(1)).get();
    String[] s=new String[0];
    Cell cs=new Cell($t[3],new $Array($t[2],s));
  }
  static $TD[] $t=new $TD[5];
  static{
    $t[0]=$TDManager.register(String.class);
    $t[1]=Cell.createTD(new $TD[]{$t[0]});
    $t[2]=$Array.createdTD(new $TD[]{$t[0]});
    $t[3]=Cell.createTD(new $TD[]{$t[2]});
    $t[4]=$Array.createdTD(new $TD[]{$t[1]});
  }
}
```

Figure 14. *An example of code and translation when using arrays*

7.2 Non-Parametric extension of parametric types

In the example of codes we showed in previous chapter, one parametric class could extend from another one and, so to speak, inherit some of its type parameters too. For instance `Pair<R,S>` extended from `Cell<R>`. Another interesting case is the one in which one class extends from an instantiated parameterized class.

Consider a class `Point` which extends from `Pair<Integer,Integer>` and defines a method that returns the distance of the current point to the origin. `Point` will certainly use both inherited methods `Cell.get` and `Pair.getSecond` as their return type were `Integer`. A client class will create class instances of `Point` as it were non-generic. Checking if such an object would be an instance of `Pair<Integer,Integer>` should return a positive result.

The handling of such a case is immediate by considering class `Point` as a generic class with zero type variables. At load-time class `Point` will create and register its type descriptor, which will have the type `Pair<Integer,Integer>` as its father type, and no friend types nor type variables. The father type should be passed to the super-constructor according to what has already been discussed. Client classes will use `Point` in instantiations as if it were non-generic, but will use the translation pattern for instance tests and casts. See Figure 15. Erasure technique leads to a complication when extending a parametric class in which one of its parameters is instantiated. In particular a mismatch occurs when trying to define a method `M` that overrides a method of the super-class, and the latter uses one of these instantiated parameters in its signature. This problem has been already highlighted by *GJ*'s authors which proposed the solution which consists in the creation for each of these methods `M` of one more bridge method `M'` which accommodate the type mismatch caused by the erasure (see also [CS98] for more details). Such a solution is directly applicable to *LM* too.

The underlying problem of the erasure technique is usually associated to the use of the bounded parametric polymorphism, that can be implemented in our framework in the same way as described in [*NextGen*].

7.3 Parametric Interfaces

Another desirable feature is to allow interfaces, that in fact are nothing but full abstract classes, to be parameterized. The main problem involved in the implementation of this feature is that in this case multiple inheritance should be provided.

First of all parameterized interfaces will have their own `$TD` objects, carrying the `Class` attribute and the type parameters, but since they have no method bodies, they will not have friend types, nor father type. They will not provide a `createTD` method but their registration will be realized directly using a new method `registerInterface` of the `$TDManager` .

```
class Point extends Pair<Integer,Integer>{
  Point(Integer x,Integer y){
    super(x,y);
  }
  int getDiagonal(){
    int x=get().intValue();
    int y=getSecond().intValue();
    return Math.sqrt(x*x+y*y);
  }
}

class iClient{
  public static void main(String[] s){
    Point P=new Point(
       new Integer(1),new Integer(2));
    boolean b=
       P instanceof Pair<Integer,Integer>;
  }
}
```

```
class Point extends Pair{
  static $TD $td=createTD();
  static createTD(){
   $TD t=$TDManager.register(Point.class);
   if (t.checkNew()){
    $TD t_1=$TDManager.register(Integer.class);
    t.father=Pair.createTD(new $TD[]{t_1,t_1});
    t.friends=$TD.voidTDs;
   }
   return t;
  }
  Point(Integer x,Integer y){
    super($td.father,x,y);
  }
  int getDiagonal(){
    int x=((Integer)get()).intValue();
    int y=((Integer)getSecond()).intValue();
    return Math.sqrt(x*x+y*y);
  }
  $TD getTD(){ return $td; }
}

class iClient{
  static $TD $t[]=new $TD[2];
  static{
    $t[0]=$TDManager.register(Integer.class);
    $t[1]=Pair.createTD(new $TD{$t[0],$t[0]});
  }
  public static void main(String[] s){
    Point P=
      new Point(new Integer(1),new Integer(2));
    boolean b=$t[1].isInstance(P);
 }
}
```

Figure 15. *Example of code and translation for non-parametric extension*

To handle subtyping in this new framework, the direct access technique results well appliable than the navigational one. In fact, with the direct access approach we can simply make the `createTD` method of each parameterized class to set in the subtyping string the appropriate bit for each interface it implements. This solution do not affect in a sensible manner neither the load-time nor the run-time performances.
An interesting issues about parametric interfaces is the difference of expressiveness of full heterogeneous translation versus homogeneous translation. The former allows a class to implement different instantiation of a unique parametric interface, while homogeneous translations are unable to realize it. The same difference hold for classes when these two approaches are applied to languages with multiple inheritance for classes.

7.4 Parametric methods

Recent contributes of extension of Java with parametric types also include the feature of parametric methods, that is methods which can abstract from the instantiation of one ore more type variables. The instantiation for these type parameters has to specified each time the parametric method is invoked. The whole mechanism is quite similar to the idea of parametric classes. In Figure 16 an example for the use of parametric methods is given. It presents a version of the `Cell` class called `MCell`, that defines some new methods. The parametric method `getMPair<R>` is used to obtain a new `MPair` object with the first component equals to the content of the receiver `MCell` object, and the second equals to the input argument. The parametric method `gather<A,B>` uses the variables `A,B,T` in its body, where `T` is the type variable of the class `MCell<T>` within it is defined. It creates some `MPair` and `MCell` objects.

```
class MCell<T>{
  T t;
  MCell(T t){ this.t=t;}
  public MPair<T,R> getMPair<R>(R r){
     return new MPair<T,R>(t,r);
  }
  public void gather<A,B>(A a,B b){
    MPair<A,B> p1=new MPair<A,B>(a,b);
    MPair<T,A> p2=new MPair<T,A>(t,a);
    MPair<MPair<A,B>,MPair<T,A>> p3=
     new MPair<MPair<A,B>,MPair<T,A>>(p1,p2);
    MPair<T,A> p4=getMPair<A>(a);
    MCell<A> c1=new MCell<A>(a);
  }
  public MPair<T,T> getDuplicate(){
    return getMPair<T>(t);
  }
}
```

```
class MPair<R,S> extends MCell<R>{
  S s;
  Pair(R r,S s){ super(r); this.s=s;}
}

class methodClient{
  public static void main(String[] s){
    MCell<Integer> c=
        new MCell<Integer>(new Integer(1));
    MPair<Integer,String> p=
        c.getMPair<String>("2");
    c.gather<Float,Long>(
        new Float(0.5),new Long(0));
  }
}
```

Figure 16. *An example of use of parametric methods*

Finally the (non parametric) method `getDuplicate()` uses a parametric method to create a `MPair` object which represents a duplication of the content of the receiver `MCell` object. The client class uses one of these methods by specifying the instantiation of the type parameters. This example should cover all the possible cases of integration between parametric methods, parametric classes, class type variables, method type variables.

To implement the translation for this new mechanism the classes library as defined in previous sections has to be modified. First of all we introduce a class named `$MD`, which is the type for the method descriptors.

A type descriptor is identified with its class and its type parameters, a method type descriptor will be identified by its defining class, a unique identifier for the method within the class (for example an `int` value), its method type parameters, the type parameters of the object which is the receiver for the invocation. For example the `$MD` object relative to the parametric method invoked by the `MClient` class is identified by the class `MCell`, the identifier 1 (since it is the first parametric method of `MCell`), the method type variables `Float` and `Long`, and the class type variable `Integer`[14].

This latter is simply obtainable if the parametric method is defined as `final`, in such case in fact the type variables of the receiver object can be statically discovered. When dealing with overriding and inheritance instead, such information is not known at compile-time, and we believe it is not possible to find an efficient implementation for this handling.

Such a problem is typical to all the approaches which statically gather type information, such *HT* and *NextGen*. The latter in particular requires a new method to be added to the class, each time a different instantiation of it is used, leading to class files which grows as time passes.

Supposing not to handle overriding for parametric methods, one can manage method type descriptors as class descriptors. They will be registered by client classes, which call a new method `createMD` analogous to `createTD` (both implemented within the generic class). Such method descriptors will be passed to parametric methods invocation, as usual as the first argument, and then used in their bodies to perform type operations over the type variables. So, both type descriptor and method descriptor will have both friend type descriptors and friend method descriptors.

The example we gave includes all these cases, in fact `MCell<Integer>` will have the friend method descriptor (field `$friendMDs`) `MCell<Integer>.getMPair<Integer>`, while `MCell<Integer>.gather<Float ,Long>` will have the friend method (field `$friendMDs`) `MCell<Integer>.getMPair<Float>` and the friend type (field `$friendTDs`) `MPair<Integer,Float>`.

`$MD` objects have to be registered to the `$TDManager` , which ensures they won't be created twice, they do not need any subtype management, they must provide `hashCode` and `equals` methods. Client classes that use them must built them at load-time, and parametric classes which instantiate them using

[14] We identify such method by writing: MCell<Integer>.gather(Float,Double)

some of its type parameters must provide a static `createMTD` methods, which behaves like the method `createTD`. The new class libraries are shown in Figures 17, 18, 19, while the translation of the source code of Figure 1 is in Figure 20 and Figure 21.

The whole management increases the overhead of the translation, but without exceeding the bounds discussed in Section 5. *GJ* uses an inferring technique which allows the avoidance of explicit instantiation of type variables during method calls, as a facility for programmers. Such technique can be potentially used in our translation too.

7.5 Bounded Polymorphism

With the parametric polymorphism we introduced so far, a programmer can more and less just design generic collection of objects. This because when one design a parametric class, it can not make any assumption about the class to which the type variables will be instantiated to, all it knows is that it is a class that extends from *Object*[15].
This lack causes not to allow a true cooperation between a generic collection and its internal elements. For these reasons type theory introduced a variation of the parametric polymorphism, called bounded polymorphism. With this type system when one defines a parametric type, it has to specify for each type variable a type, called bound, which has to be super-type for all its possible instantiations.
In Java this has been mapped allowing to specify for each type variable which class its instantiation have to extend or which interface to implement. Compile-time checks ensure that the instantiations of a parametric class satisfy all the bounds. In heterogeneous translations that is enough to support this kind of parametric polymorphism, but in homogeneous translations the erasure technique have to be changed a little. In particular all the uses of a type variable within its class definition, are not translated straightforward to `Object` but to the bound, or in the case it is parametric too, to its erasure. The general implementation of this mechanism in our translator do not change with respect to that realized with *GJ* or *NextGen*. See Figure 22.

8 Implementation Issues

8.1 Security

Homogeneous translations erase type information at compile-time, so they causes some problems when this information is needed to be used at run-time. The usual example to highlight this is the one given in [AFM97], in which a client class not written with the translator uses a method which accepts an argument of type `List<SecureChannel>`, by passing it a list of (non-secure) `Channel` elements[16].
In [OW98] it has been proposed to implement such critical classes by borrowing an heterogeneous translation technique, that is by using a class `ListofSecureChannel` which extends from `List<SecureChannel>`. This guarantees the content of the list to be type-consistent, but needs the creation of the usual redundant code.
With our approach the same technique can be used, but it is also possible to perform type consistency controls at run-time, by adding secure setter methods to the list which explicitly control the conformance of the input arguments with respect to the instantiation of the type variables. See Figure 23. This new code may be automatically added by the translator each time the programmer specifies that a method needs such a security management, for example using a new method modifier called *secure*. Such a technique obviously leads to a little of run-time overhead.

8.2 Compatibility

In [OW98] a new issue has been addressed concerning the translation approach to extension of Java with parameterized types. Typically an extension obtained through translation has the backward compati-

[15]As for previous homogeneous translations (and as for NextGen), we do not admit primitive types to be instantiation for type variables, but the translator automatically maps them into their correspondents wrappers (Integer for int, Float for float, ..)
[16]Here SecureChannel extends from Channel implementing security checks when a channel is created

```
public class $TD {
    public int ID;

    public Class c;
    public $TD[] params;
    public $TD[] friendTDs;
    public $MTD[] friendMDs;
    public $TD father;

    private int hashValue;
    private boolean bNew;

    public static $TD[] voidTDs;
    public static $TD objTD;

    static {
        objTD=new $TD(Object.class,voidTDs);
        objTD.ID=0;
        objTD.bNew=true;
        objTD.father=objTD;
    }

    public $TD(Class c, $TD[] p) {
        this.c=c; params=p; bNew=true;
        friendTDs=null; friendMTDs=null;
        father=null; ID=-1;
        processHash();
    }
    public boolean checkNew() {
        if (!bNew) return bNew;
        bNew=false;
        return true;
    }
    public boolean equals(Object obj) {
        if (!(obj instanceof $TD)) return false;
        $TD S=(($TD)obj);
        if (!S.c.equals(c)) return false;
        if (S.params.length!=params.length) return false;
        for (int i=params.length-1;i>=0;i--)
            if (S.params[i].ID!=params[i].ID) return false;
        return true;
    }
    public int hashCode() {
        return hashValue;
    }
    public boolean isInstance(Object o) {
        return o instanceof $Parametric
            ?navSubtype((($Parametric)o).getTD(),ID)
            :c.isInstance(o);
    }
    public static final boolean navSubtype($TD P,int pos) {
        int Q;
        while ((Q=P.ID)!=pos && Q!=0) P=P.father;
        return Q==pos;
    }
    void processHash() {
        hashValue=c.hashCode();
        for (int i=params.length-1;i>=0;i--)
            hashValue+=params[i].ID;
    }
}
```

Figure 17. *Implementation of class $TD handling parametric methods*

```
public class $MD {
    public int ID;
    public Class c;
    public int methID;
    public $TD[] classParams;
    public $TD[] methodParams;
    public $TD[] friendTDs;
    public $MD[] friendMTDs;

    private int hashValue;
    private boolean bNew;

    public static $MD[] voidMTDs=new $MD[0];

    public $MD(Class c, int methID,$TD[] classP,$TD[] methP) {
        this.c=c; this.methID=methID; this.classParams=classP;
        this.methodParams=methP; bNew=true; friendTDs=null;
        friendMTDs=null;    ID=-1;
        processHash();
    }
    public boolean checkNew() {
        if (!bNew) return bNew;
        bNew=false;
        return true;
    }
    public boolean equals(Object obj) {
        if (!(obj instanceof $MD)) return false;
        $MTD S=(($MD)obj);
        if (!S.c.equals(c)) return false;
        if (S.methID!=methID) return false;
        if (S.classParams.length!=classParams.length) return false;
        for (int i=classParams.length-1;i>=0;i--)
            if (S.classParams[i].ID!=classParams[i].ID) return false;
        if (S.methodParams.length!=methodParams.length) return false;
        for (int i=methodParams.length-1;i>=0;i--)
            if (S.methodParams[i].ID!=methodParams[i].ID) return false;
        return true;
    }
    public int hashCode() {
        return hashValue;
    }
    void processHash() {
        hashValue=c.hashCode()+methID;
        for (int i=classParams.length-1;i>=0;i--)
            hashValue+=classParams[i].ID;
        for (int i=methodParams.length-1;i>=0;i--)
            hashValue+=methodParams[i].ID;
    }
}
```

Figure 18. *Implementation of class $TDM*

```
public class $TDManager {
    private static Hashtable hTd;
    public static int countIDTd;
    private static Hashtable hMd;
    public static int countIDMd;
    static{
        hTd=new Hashtable(101,0.7f);
        hTd.put($TD.objTD,$TD.objTD);
        countIDTd=1;
        hMd=new Hashtable(101,0.7f);
        countIDMtd=0;
    }

    static $MD registerMD(Class c,int methID,$TD[] classP,$TD[] methP) {
        $MD S=new $MTD(c,methID,classP,methP);
        Object o=hMd.get(S);
        if (o==null){
            hMd.put(S,S);
            S.ID=countIDMd++;
            return S;
        }
        return(($MD)o);
    }
    static $TD registerTD(Class c) {
        $TD S=new $TD(c,$TD.voidTDs);
        Object o=hTd.get(S);
        if (o==null){
            hTd.put(S,S);
            S.ID=countIDTd++;
            S.father=registerTD(c.getSuperclass());
            return S;
        }
        return(($TD)o);
    }
    static $TD registerTD(Class c,$TD[] p) {
        $TD S=new $TD(c,p);
        Object o=hTd.get(S);
        if (o==null){
            hTd.put(S,S);
            S.ID=countIDTd++;
            return S;
        }
        return(($TD)o);
    }
}
```

Figure 19. *Implementation of class $TDManager handling parametric methods*

```
public class MCell implements $Parametric {
    static $TD $cl;
    static{
        $cl=createTD(new $TD[]{$TDManager.registerTD(String.class)});
    }

    private $TD $td;
    private Object t;
    MCell($TD $td,Object t){ this.$td=$td; this.t=t;}

    public static $MTD createMTD(int methID,$TD[] classParams,$TD[] methParams) {
        $MD t=$TDManager.registerMTD(MCell.class,methID,classParams,methParams);
        if (t.checkNew()){
          if (methID==0){
            t.friendTDs=new $TD[1];
            t.friendTDs[0]=MPair.createTD(new $TD[]{classParams[0],methParams[0]});
            t.friendMDs=null;
          } else if (methID==1){
            t.friendTDs=new $TD[4];
            t.friendTDs[0]=MPair.createTD(new $TD[]{methParams[0],methParams[1]});
            t.friendTDs[1]=MPair.createTD(new $TD[]{classParams[0],methParams[0]});
            t.friendTDs[2]=MPair.createTD(new $TD[]{t.friendTDs[0],t.friendTDs[1]});
            t.friendTDs[3]=MCell.createTD(new $TD[]{methParams[0]});
            t.friendMDs=new $MD[1];
            t.friendMDs[0]=MCell.createMD(0,classParams,new $TD[]{methParams[0]});
          }
        }
        return t;
    }
    public static $TD createTD($TD[] params) {
        $TD t=$TDManager.registerTD(MCell.class,params);
        if (t.checkNew()){
            t.father=$TD.objTD;
            t.friendTDs=null;
            t.friendMDs=new $MD[1];
            t.friendMDs[0]=createMD(0,params,params);
        }
        return t;
    }
    public void gather($MD $md,Object a,Object b) {
        MPair p1=new MPair($mtd.friendTDs[0],a,b);
        MPair p2=new MPair($mtd.friendTDs[1],t,a);
        MPair p3=new MPair($mtd.friendTDs[2],p1,p2);
        MPair p4=getMPair($mtd.friendMDs[0],a);
        MCell c1=new MCell($mtd.friendTDs[3],a);
    }
    public MPair getDuplicate() {
        return getMPair($td.friendMDs[0],t);
    }
    public MPair getMPair($MTD $md,Object r) {
        return new MPair($md.friendTDs[0],t,r);
    }
    public $TD getTD() {
        return $td;
    }
}
```

Figure 20. *Translation of class MCell*

```
public class MPair extends MCell {
    private $TD $td;
    Object s;

    public MPair($TD $td,Object r,Object s) {
        super($td.father,r);
        this.$td=$td;
        this.s=s;
    }
    public static $TD createTD($TD[] params) {
        $TD t=$TDManager.registerTD(MPair.class,params);
        if (t.checkNew()){
            t.father=MCell.createTD(new $TD[]{params[0]});
            t.friendTDs=null;
            t.friendMDs=null;
        }
        return t;
    }
    public $TD getTD() {
        return $td;
    }
}

public class MClient {

    static $TD[] $t=new $TD[6];
    static $MD[] $m=new $MD[2];
    static{ init();}

    public static void init() {
        $t[0]=$TDManager.registerTD(Integer.class);
        $t[1]=MCell.createTD(new $TD[]{$t[0]});
        $t[2]=$TDManager.registerTD(String.class);
        $t[3]=MPair.createTD(new $TD[]{$t[0],$t[2]});
        $t[4]=$TDManager.registerTD(Float.class);
        $t[5]=$TDManager.registerTD(Long.class);
        $m[0]=MCell.createMD(0,new $TD[]{$t[0]},new $TD[]{$t[2]});
        $m[1]=MCell.createMD(1,new $TD[]{$t[0]},new $TD[]{$t[4],$t[5]});
    }

    public static void main(java.lang.String[] args) {
        MCell c=new MCell($t[1],new Integer(1));
        MPair p=c.getMPair($m[0],"2");
        c.gather($m[1],new Float(0.5),new Long(0));
    }
}
```

Figure 21. *Translations of classes MPair and MClient*

```
class NumberCell<T implements Number> {
  T t;
  NumberCell(T t){this.t=t;}
  NumberCell<Integer> getIntCell(){
    return new NumberCell<Integer>(
        new Integer(t.intValue()));
  }
}
```

```
class NumberCell {
 private $TD $t=
   createTD($TDManager.register(Integer.class));
 private $TD $td;
 Number t;
 NumberCell($TD $td,T t){
    this.$td=$td; this.t=t;}
 NumberCell getIntCell(){
   return new NumberCell($t,
            new Integer(t.intValue()));
 }
}
```

Figure 22. *Bounded Polymorphism: example and translation*

```
class List<T>{ ...
  private void nonSecureSet(T t){...}
  public void secureSet(Object t){
    if (t instanceof T) nonSecureSet((T)t);
  }
}
```

Figure 23. *Preventing security attacks using dynamic constrols*

bility property, that is the translator is able to accept a standard Java code, and does not change its behaviour due to the translation[17]. This property holds for *Pizza*, *GJ*, *NextGen* and for *LM* too. A different property is the forward compatibility, that is the possibility for a standard Java code written to work with standard libraries (legacy code), to be able to work in the same way when such libraries are substituted with their generic version. In particular such a situation becomes useful when considering the benefits of the upgrading of the collection library of Java ([JCOLL]).
In order for this property to hold, the translation should be able to create classes whose signature is the same as the one obtained using the generic idiom, in particular it must rely on the erasure technique. This is done well in *GJ*, and requires some glue code to work with *NextGen*.
The translation as we have defined it so far does not have this kind of compatibility due to a set of changes on the classes signature: constructors provide one more argument, parametric classes implement one more interface, parametric classes have two new public methods and a new private field, client classes have some new private and static fields. The private properties of a class do not cause problems, since their presence cannot be perceived by other classes. Furthermore we believe that the introduction of public properties with mangled names [18] can be considered acceptable to the end of forward compatibility, as it can be discovered only by asking for all the properties of a generic class using a reflective inspection. The only critical point remains the first. However a little change on the translation can accommodate it, with just a little loss in efficiency. The constructor of a parametric class can be translated so that it does not require the type descriptor as first argument, but just stores in the `td` field a default descriptor corresponding to all the variables instantiated to their bounds[19]. To change the `td` field a new method "`void $init($TD td)`" must be used . Each constructor invocation is then translated into a sequence of a *new* operation and *init* method call. In order to support such a technique we have to force a constructor not to perform type operations using current instantiation of type variables.
Legacy code, which just calls the constructor and does not invoke *$init* method, obtains objects with the same semantics as that of generic idiom, obtaining the forward compatibility.

[17] More likely in this case, translations do not change the code at all

[18] We consider a mangled name a name which contains some special character that programmers typically do not use in their programs, such as "$". Notice that this technique has already been used by other translations such as NextGen, and by Java designers to implement inner classes

[19] Such a descriptor can be built at load-time and stored into a new static field

```
\\Source code

class Cell<T> implements Serializable{
 ...
}

class Client1{  // Storing an object
 void storeAnObject() throws IOException{
   Cell<Integer> c=
        new Cell<Integer>(new Integer(2));
   FileOutputStream f=
        new FileOutputStream("c:\store.dat");
   ObjectOutputStream o=
        new ObjectOutputStream(f);
   o.writeObject(c); o.close();
 }
}

class Client2{ // Reading the object
 void getTheObject() throws
        IOException, ClassNotFoundException{
   FileInputStream f=
        new FileInputStream("c:\store.dat");
   ObjectInputStream i=
        new ObjectInputStream(f);
   Object obj=i.readObject();
   i.close();
   ...
   Cell<Integer> c=(Cell<Integer>)obj;
 }
}
```

```
\\Translation

class Cell implements Serializable{
 ...
 private void readObject(ObjectInputStream i)
     throws IOException,ClassNotFoundException{
    i.defaultReadObject();
    $td=$TDManager.registerAgain($td);
 }
}

class $TD{ // A change on the $TD implementation
 ...
$TD registerAgain($TD $td){
  if (!$Parametric.class.isAssignableFrom($td.c))
    return register($td.c);
  $TD at[]=new $TD[$td.params.length];
  for (int i=0;i<at.length;i++)
    at[i]=registerAgain($td.params[i]);
  $TD t=register($td.c,at);
  if (t.checkNew()) {
    t.father=registerAgain($td.father);
    t.friends=new $TD[$td.friends.length];
    for (int i=0;i<$td.friends.length;i++)
     t.friends[i]=registerAgain($td.friends[i]);
  }
  return t;
} }
```

Figure 24. *Intercepting the serialization mechanism*

8.3 Persistence

Java allows persistence using the core Serialization mechanism. By making a class implement the interface
`java.io.Serializable`, objects belonging to that class may be directly written into a data stream, and subsequently read, possibly by another application. This allows objects to be transferred through the net (using socket-stream based communications) or to be stored in the file system (using file streams). Java also permits the writing of its own code to write and read objects, by defining methods `writeObject` and `readObject`.

In order to allow new applications to be able to get the stored object, the class loader should retrieve its class file[20] so that a new object with the same content as the previous can be created. For more information about the serialization mechanism of Java see [JSer].

Obviously we would require our translation to allow an object created from a parametric type to be written and subsequently read by another application. With this respect our main concern is that its type descriptor can be, so to speak, "adapted" to the `$TDManager` static environment of the application which will read it. A type descriptor carries some information depending on the status of the `$TDManager` when it was registered, like its unique identifier. This means that when a type descriptor needs to be imported into a new application it needs also to be registered again in order to achieve a proper integration in the new environment.

See the code of Figure Ser. As one may notice there is no need for the translation to affect in any way (but those already discussed) client classes. It just redefines the method `readObject` in order to accommodate the type descriptor in the new run-time environment. This is realized by invoking the `registerAgain` functionality when a parametric object is read.

[20] When such a class is not found a java.lang.ClassNotFoundException is raised

In the example of Figure 24 we suppose that the programmer has not redefined `readObject` itself, hence the translator used the method `defaultReadObject` to call the default implementation of the serialization. If the contrary is true, and the programmer has already give an implementation of `readObject`, then the translator inserts the `registerAgain` invocation at the method `readObject` start.
The run-time cost of such an invocation is comparable to that of a first registration. Considering that an object transfer through a stream typically involves slow operating system operations, the impact of the translation is again not significant.

9 Conclusions

This paper shows how homogeneous translations are not less powerful and flexible than heterogeneous. The use of Core Java Reflection, introduced from release 1.1 of Java, plays a crucial role in this statement, as it finally allows the carrying of run-time information on instantiation. The idea of shifting the computational overhead needed for its management from run-time to load-time, allows this translation to be used without great influence on the overall performance.
Some minor drawbacks are also present. It is not already clear how to implement in an efficient way parametric methods when they should provide overriding. Furthermore the set of accepted programs is smaller than that of *RM*, due to the recursive types generation problem. All these problems seem to be, so to speak, inherited from heterogeneous approaches, since they are caused by the idea of the management of the instantiation of parametric types at load-time.
Implementation of minor issues like static members and inner classes has not been treated in this paper, but it is quite direct and it does not cause particular pitfalls.
In the cases in which the full integration between parametric types and Java types is not required, GJ remains the better solution, since it does not cause significant overhead. Such an integration can be obtained both with *NextGen* and *LM*. The former leads to higher load-time overhead and space overhead, the latter to higher run-time overhead. The contribute of this paper to the study of the introduction of parametric polymorphism in Java, is to show that with an homogeneous translation, such an integration can be achieved at acceptable costs.
It is interesting to highlight that *LM* can be considered a mixture of previous approaches: it uses the erasure technique as in *GJ*, it collects type information as in *NextGen* but in a homogeneous framework, and it redefines the class loading as in *PolyJ* but by using a translation approach.
As future work we will study the implementation of both co-variant and contra-variant type variables; *LM* and *RM* seem in fact to be the only approaches to allow the implementation of both. We are also interested in the use of core Java reflection within homogeneous approaches to translation applied to other language mechanisms, such as multiple inheritance, whose translation is currently under study.

References

[AFM97] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding parametrized type to Java. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 215-230, 1997.

[APW99] Atsushi Igarashi, Benjamin Pierce, Philip Wadler. Featherweight Java, a minimal core calculus for Java and GJ. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 132-146, 1999.

[BOW98] Kim B. Bruce, Martin Odersky, and Philip Wadler. A statically safe alternative to virtual types. *European Conference on Object-Oriented Programming*, July 1998.

[Coo89] William R.Cook. A proposal for making Eiffel type-safe. *Euopean Conference on Object Oriented Programming*, pages 52-72, October 1989.

[CS98] Corky Cartwright and Guy Steele. Compatible genericity with run-time types for the Java programming language. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 201-215, 1998.

[DGLM95] M.Day, R.gruber, B.Liskov, and A.C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 156-158, 1995.

[GS97] J.Gosling and B.Joy and G.Steele. The Java Language Specification, *Addison-Wesley*, 1996

[JCOLL] Java Collection Library Tutorial, Sun Mycrosystems, Inc.,
http://java.sun.com/docs/books/tutorial/collections/index.html

[JCR] Java Core Reflection Tutorial, Sun Mycrosystems, Inc.,
http://java.sun.com/docs/books/tutorial/reflect/index.html

[JDK1.1] The Java Development Kit 1.1, Sun Mycrosystem, Inc.,
http://www.javasoft.com/products/jdk/1.1/index.html

[JDK1.2] The Java Development Kit 1.2, Sun Mycrosystem, Inc.,
http://www.javasoft.com/products/jdk/1.2/index.html

[JSer] The Java Object Serialization Specification, Sun Mycrosystem, Inc.,
http://java.sun.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html

[Mey86] Bertrand Meyer, Genericity versus inheritance, *Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1986.

[MBL97] Andrew C. Meyers, Joseph A. Bank, and Barbara Liskov, Parameterized types for Java. *Symposium on Principles of Programming Languages*, pages 132-145, ACM, 1997.

[OW97] Martin Odersky and Philip Wadler, Pizza into Java: Translating theory into practice. *Symposium on Principles of Programming Languages*, pages 146-159, ACM, 1997.

[OW98] Martin Odersky, Philip Wadler, Gilad Bracha, and David Stoutamire, Making the future safe for the past: Adding Genericity to the Java Programming language. *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 183-200, 1998.

[SA98] Johe H. Solorzano and Suad Alagic, Parametric Polymorphism for Java: A Reflective Solution, *Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 216-225, 1998.

[Tho97] Kresten Krab Thorup, Genericty in Java with virtual types. *European Conference on Object-Oriented Programming*, pages 444-471, LNCS 1241, Springer-Verlag, 1997

[Tor98] Mads Togersen. Virtual types are statically safe. *5th Workshop on Foundations of Object-Oriented Languages*, January 1998.