Cooking Raw Types in Java

Dmitri Boulytchev[1] and Eugene Vigdorchik[2]

[1] St.Petersburg State University, Universitetskii pr., 28
198504, St.Petersburg, Russia,
db@tepkom.ru,
[2] JetBrains.com, Kantemirovskaya ul., 2,
197342, St.Petersburg, Russia
ven@intellij.com

Abstract. Recently released version 5.0 of the Java programming language introduces among other new language features the construct of generic classes. This language extension allows the classes to declare type parameters and the users to instantiate those classes giving explicit types as values for those parameters. To facilitate migration to Java 5.0, existing legacy code using parameterless instantiations is still valid with the new compiler, though this does not guarantee type safety. We discuss the problem of automatic conversion of existing code to use generic types and present a technique based on type inference. The evaluation of the approach performed on some real-world industrial projects shows that it is superior to previously reported approaches in reliability while also delivering competitive quality and performance.

1 Introduction

For many years Java programmers have been enjoying the ability to exploit the numerous libraries bundled with the compiler. Probably the most frequently used of those is the Collections framework, that saved programmers from reimplementing basic data structures. However, due to the weakness of the Java type system, collections were declared to contain any objects, and users were forced to cast objects from the collection to the desired type, essentially undermining type safety of the program.

In Java 5.0 *generic*, or parameterized, classes are introduced to cope with this shortcoming. Generic class may declare type parameters that have the scope of this class; a class may have more than one parameter. Generic class may then be instantiated for various actual types of its parameters giving the compiler the opportunity to effectively typecheck on a per-instantiation basis.

For compatibility reasons any generic class may be used in non-parameterized, or *raw*, form. In this case all its type parameters are considered to be bound to some virtual "raw argument" type, hereafter referred to as a "bottom type" ($\perp$). This type does not have any representation in Java and may appear only as an invisible argument of raw type instantiation.

Raw types allow to use all of the legacy non-generic code together with reworked generic standard libraries without conversion. On the other hand, conversion is desirable since generic code has a more rigorous type structure and exposes more clear interfaces. In this paper we present a technique for automatic refactoring of non-generic code to a generic equivalent. This technique appeared to be tractable and has been implemented as a dedicated refactoring in the IntelliJ IDEA 5.0[3] development environment.

2 Generic and Raw Types, Generification and Cooking

Throughout this section we give an overview of generic classes and types in Java 5.0 and discuss various aspects of converting existing code to generic-aware form.

To illustrate the need for generic classes consider for example the following imaginary set declaration:

```
class Set {
   void    put      (Object value) {...}
   boolean contains (Object x    ) {...}
   Object  anyOf    (            ) {...}
}
```

The client of the `Set` class who knows to put only `Integer`s in the set, but still has to cast the result returned by `anyOf` call to `Integer` which, if the user is wrong, results in a runtime exception. It seems reasonable to allow the user to specify explicitly what kind of objects are stored in the collection, giving the compiler more knowledge about the program and making the above cast redundant.

In Java 5.0 the above class definition would be rewritten in the following manner:

```
class Set<X> {
   void    put      (X x) {...}
   boolean contains (X x) {...}
   X       anyOf    (   ) {...}
}
```

Here `Set` is declared as a generic class with one type parameter `X`. A generic class may be instantiated for various actual types of its parameters in the following manner:

```
Set<Integer> s = new Set<Integer> ();
s.put (new Integer(3)); // Ok
s.put (Boolean.TRUE);   // type error
```

[3] http://www.jetbrains.com/idea

The variable `s` is declared to be of type `Set<Integer>`; now the compiler is able to enhance the type checking.

To provide compatibility with legacy non-generic code non-parameterized form of generic types is also available:

```
Set s = new Set ();
s.put (3);    // Ok
s.put (true); // Ok
```

Here the variable `s` has the type "raw" `Set`, or `Set` parameterized by virtual "bottom" type ($\perp$); since any reference type can be converted to bottom type, this code is type-safe.

Moreover, it is allowed, though discouraged, to mix generic and raw types:

```
Set<String> strings = s; //compiler warning
String str = strings.anyOf(); //OK at compile time, but exception at runtime
```

Here a special "unchecked" warning is issued by the compiler, signalling that the program is not guaranteed to be runtime safe. Once there are no unchecked warnings, the program is guaranteed to produce no cast exceptions when evaluating expressions of types of generic classes.

An important feature of generic type system, that is sometimes missed by the users, and which clearly affects generification, is the invariance of types with regard to their type arguments, i.e. `Set<Integer>` is *not* a subtype of `Set<Object>` even though `Integer`<:`Object`.

Local type inference is another feature that has to be taken into account: in Java 5.0 the type of expression may generally depend not only on types of its subexpressions, but also on the type of the outer context. For example, in the following example

```
<T> List<T> emptyList () {...}

List<String> s = emptyList ();
List<Integer> s = emptyList ();
```

the return type of the first call to `emptyList` is `List<String>` while the second call returns `List<Integer>`. This example also demonstrates the presence of *generic methods* — the methods with their own type parameters.

Several aspects of converting legacy code to use generics exist. For example one may try to rewrite the code by introducing type parameters to methods and classes. In this paper we consider a simpler problem — the problem of "cooking" raw declarations to their parameterized form by introducing safe type parameters instantiation. From now on we will use the terms "generification" and "cooking" interchangeably.

A possible function to measure the results of generification is the number of unchecked conversions removed; unfortunately in some cases this goal can not

be achieved by simply adding type arguments for raw declarations. For example, `Collections` class contains raw field `EMPTY_LIST` to maintain compatibility with previously written code. To remove unchecked conversions one will need to replace occurences of that field with its generic equivalent — the call to the generic method `emptyList ()`.

Another good criterion for generification is the number of removed casts. Since a cast may be safely removed if the type of cast expression is a subtype of cast type, the number of removed casts reflects the "distance" between cooked and expected types of declarations.

Finally one may try to generify as much declarations as possible; however this intention sometimes may result in meaningless typing; for example given the declaration

```
List x;
```

that is never used the pragmatic considerations tells us to leave this declaration raw since no information is available about the type of its argument; however formally speaking it may be cooked to, say, `List<Object>` and so provide better results.

3 Related Work

The concept of generics traces its roots back to the works of [7, 1], and the earlier works on virtual types[6]. The final specification of Java 5.0 language was recently published[5]. This final standard also introduces wildcard types, a restricted form of existentials, which if incorporated into the cook process is likely to increase the number of generified items.

The refactoring of non-generic legacy code to use generics has been adressed by a number of authors. In [3, 8] the problem of deriving type constraints from the program is addressed. Unlike our work based on type constraints, authors seek for a solution to the points-to analysis for the generified items, requiring context-sensitive version for a more precise inference, and obtain the typing from the computed sets. The implementation described, however, is able to generify only the usages of standard Collections classes; the restriction not imposed by our refactoring.

Dincklage and Diwan[2] address the problem of simultaneous conversion of non-generic *classes* and their instantiations. While this task is more ambitious than ours, the pragmatic value of automatically adding type parameters to the user code is arguable.

Duggan's work [4] resembles ours, in the sense it is also based on successively simplifying type constraints. However the author deals with a specific subset of Java and provides no results, so no comparison is possible.

The first industrial implementation of "Generify" refactoring was done in CodeGuide IDE by OmniCore[4]. However they provided no details of the algorithm, and the empirical study of this refactoring shows poor results for both

[4] www.omnicore.com

the little number of raw items generified, and producing incompilable code. The next implementation appeared in IntelliJ IDEA 4.0, but that one used an ad hoc graph representation of constraints, and, in general, did not perform well either.

The recently released Eclipse 3.1[5] seems to provide the best results of all aforementioned works, so we measure the results of applying proposed algorithm against their implementation.

4 Setting the Scene

We consider each raw type as a generic type parameterized by *type variables*. The actual values for the type variables may then be deduced by taking into account various constraints that the program introduces on them. Informally speaking each constraint expresses the type dependency caused by a certain fragment of the cooked program. Given a set of such constraints we then resolve it using a properly developed system of inference rules. As a result we obtain a set of type variable assignments (hereafter called *binding*). As we will see shortly, such a constraint system may actually be reduced to many different bindings. Several issues are important for usability of the approach:

- all possible solutions of the constraint system must denote type-correct variable assignment;
- the inference algorithm has to be computationally tractable;
- the algorithm has to allow fine-tuning to infer bindings with various desirable properties.

Now we discuss some approaches to build the constraint system and infer solution bindings.

One trivial kind of dependency is the *subtype relation* ($<:$). For example, given an assignment one would expect that the type of its source is a subtype of the type of its destination. Thus for the construct `src = dst`, constraint $typeOf(dst) <: typeOf(src)$ has to be added to the constraint system. Similar subtyping constraints need to be introduced by method invocations and other constructs. However it turns out that subtyping relation is not sufficient to describe all available safe typings of the program.

For example, we need to take overriding of methods during inheritance into account. Given the declarations

```
class A {
   void f (Set x) {...}
}
class B extends A {
   void f (Set y) {...}
}
```

[5] www.eclipse.org

we cannot actually cook types of `x` and `y` independently — in other case we may, for example, cook type of `y` into `Set<Integer>` and leave `x` raw, in which case method `B.f` will stop override `A.f`.

Another issue is the accessibility of declarations. Consider the following example:

```
public class Escaping {
    static List x;
}

class Scope {
    private class Woof {...}

    void foo() {
        Escaping.x.add(new Woof());
    }
}
```

Since class `Woof` is not accessible at the declaration site of `x`, it is not possible to convert this declaration into generic form.

Yet another problem arises from the fact that there are no partially-raw types in Java. This means that for classes with multiple type parameters, if during cooking one of them falls into bottom, then all others have to do so. In the following example

```
Hashtable x;
List y;

y.add(x.elements().nextElement());
y.add("");
```

neither `x`, nor `y` can be cooked since the first argument of `x` is left unbound.

Another important point is dealing with casts. On one hand a type cast expression `(T) expr` only succeeds at run time if the type of `expr` appears to be a subtype of `T`; from this point of view one may conclude that the cast would introduce the constraint $typeOf(expr) <: T$. On the other hand, it is enough for $typeOf(expr)$ to be *convertible* [5] to T for the cast to be type-correct at compile time. Convertibility is a weaker constraint than the subtyping and so taking it into account may result in different typing. Consider the following example:

```
List x;
String s = (String) x.iterator().next();
Integer i = (Integer) x.iterator().next();
```

If we treat these two casts as subtype constraints then no typings exist since there is no type that is simultaneously subtype for both `String` and `Integer`. On the other hand at least three types (`Object`, `Cloneable` and `Serializable`)

are convertible to both `String` and `Integer`. This example seems a bit artificial, but a similar situation may appear when, for example, the same iterator is used to access collections with different possible parameterizations.

5 Inferring the Types

In this section we informally describe the type inference system used in our approach. This system allows us to infer type-safe parameterizations for some restricted version of Java 5.0 type system. The main restrictions are as follows:

- we consder cast expression to contribute subtyping constraint on type of casted expression rather than convertable constraint;
- we do not allow assignment and method invocation conversions[5] of raw type to parameterized type.

The idea behind our approach to generification is quite simple: given a set of constraints, we try to reduce it by choosing arbitrary ones and building a local solution for it; then we combine local solutions into a global one. Since many different solutions may exist for the same constraint system, we generally have non-deterministic type inference rules that may deliver different local solutions for the same constraint. The latter means that we have, in general, an *inference tree*; the root of which corresponds to the initial system, interior nodes correspond to partial solutions, and leaves relate to final solutions or contradictions (some local solutions may contradict others, in which case the inference process stops.) If any correct typing corresponds to some solution of the constraint system, and if type inference rules are able to infer any solutions, then we may enumerate all typings and choose any desirable one. Of course this very approach seems to be intractable due to exponential growth of the number of solutions; however it turns out that even for large projects (up to 100000 lines of code) it delivers results in acceptable time; we also developed a simple heuristic that dramatically reduces the running time.

Since the formal description of the type inference system is outside the scope of this paper, we demonstrate the basic rules with the following example:

```
List x;
List y;
List z;

x.add(y);
z = (List) x.iterator().next();

y.add("");
```

The constraint system for the example is as follows:

$$
\begin{gathered}
\mathit{List}\,\langle\gamma\rangle <: \mathit{List}\,\langle\delta\rangle \\
\mathit{List}\,\langle\beta\rangle <: \alpha \\
\alpha <: \mathit{List}\,\langle\delta\rangle \\
\mathit{String} <: \beta
\end{gathered}
$$

Here the type variables α, β, γ, δ correspond to type parameters of `x`, `y`, `z` and cast expression respectively.

Now we can start reducing this system. First of all we choose the first constraint $\mathit{List}\,\langle\gamma\rangle <: \mathit{List}\,\langle\delta\rangle$. Generally speaking there are an infinite number of solutions for this constraint, but all solution space may be represented by two relations on type variables γ and δ. The first relation says that δ and γ have to denote the same type. We may express this fact by substitution $\delta \to \gamma$. The second says that if γ is bound to bottom type ($\gamma \to \bot$) then the constraint is satisfied regardless the binding of δ. In other words we may replace the constraint under considerations by one of two bindings ($\delta \to \gamma$ or $\gamma \to \bot$). These bindings form first two possible local solutions. Now we may replace the initial system with *two* derived ones: they are obtained by removing the first constraint and applying the corresponding variable substitutions to types of other constraints. These two sysems are shown below:

$$
\begin{gathered}
\delta \to \gamma \\
\mathit{List}\,\langle\beta\rangle <: \alpha \\
\alpha <: \mathit{List}\,\langle\gamma\rangle \\
\mathit{String} <: \beta
\end{gathered}
$$

and

$$
\begin{gathered}
\gamma \to \bot \\
\mathit{List}\,\langle\beta\rangle <: \alpha \\
\alpha <: \mathit{List}\,\langle\delta\rangle \\
\mathit{String} <: \beta
\end{gathered}
$$

Consider the first two constraints of the first system: $\mathit{List}\,\langle\beta\rangle <: \alpha$ and $\alpha <: \mathit{List}\,\langle\gamma\rangle$. Clearly they both may be satisfied simultaneously iff $\mathit{List}\,\langle\beta\rangle <: \mathit{List}\,\langle\gamma\rangle$. Again, we have two choices: $\beta \to \gamma$ or $\gamma \to \bot$. The first of them results in the system

$$
\begin{gathered}
\delta \to \gamma \\
\beta \to \gamma \\
\alpha \to \mathit{List}\,\langle\gamma\rangle \\
\mathit{String} <: \gamma
\end{gathered}
$$

and the second results in the system

$$
\begin{gathered}
\delta \to \bot \\
\gamma \to \bot \\
\mathit{List}\,\langle\beta\rangle <: \alpha \\
\mathit{String} <: \beta
\end{gathered}
$$

Note that we apply the local solution not only to the residual system, but also to a partial solution.

Again consider the system derived first at the previous step. Here we have only the constraint $String <: \gamma$. The only possible solutions here are $\gamma \to String$, $\gamma \to Comparable$, $\gamma \to Serializable$ or $\gamma \to Object$; the most promising is of course the first one. Applying it to the partial solution we have the binding

$$\begin{gathered} \delta \to String \\ \beta \to String \\ \alpha \to List\,\langle String \rangle \\ \gamma \to String \end{gathered}$$

that represents the first possible solution of the initial system. The other solutions may be deduced by considering other branches of the inference tree.

Now we may summarize the main steps of the inrefence algorithm:

- We maintain the partial solution and residual system. Initially the partial solution is empty and the residual system is taken to be the one obtained from the program at previous step.
- At each step we try to "pump" some constraints from the residual system into the partial solution: we describe all possible solutions of these separated constraints by set of disjoint bindings and then incorporate these bindings into the partial solution.
- When we end up with an empty residual solution, then partial solution is the solution of the initial system; if none of the constraints may be solved then we give up.

The type inference system developed allows to obtain all possible solutions of the system regardless of the order of rule application. However the compexity of this algorithm is exponential to the number of constraints. On many relatively small projects (up to 10 000 lines of code) it is still appropriate, while generification of large projects (up to 100 000 lines of code) may take up to ten hours. Fortunately we have found a simple heuristic that allows us to dramatically reduce the complexity of the algorithm.

As one may notice one of the main sources of exponential growth is the rule for solving constraint of the form $C\,\langle\alpha\rangle <: C\,\langle\beta\rangle$, because we have to consider two bindings $\alpha \to \beta$ and $\beta \to \bot$. Our heuristic allows us to drop the latter solution in most cases. Namely, we drop it if variable β does not occur elsewhere in the residual system. Experimenting shows that using this heuristic we lose no more than 10 percents of generified items while running time becomes comparable to that reported in previous works. All results reported in the following section are given with respect to this heuristic.

6 Implementation and Results

We measured the obtained results against all results reported in papers earlier, and found our implementation superior to them, both producing more generic

items, and eliminating more casts. Recently released Eclipse 3.1 framework, however, also provides best results in comparison to that papers; despite the absence of exact details on algorithm used by Eclipse we compared our results with it.

In table 1 the results of the application of our algorithm and the "Infer type arguments" refactoring presented by Eclipse [9] are given for benchmarks used by other authors. The comparison shows that the Eclipse implementation usually outperforms ours in terms of time required to perform the refactoring. This can be explained by the context-sensitive nature of the algorithm they use [8]: having the points-to information should allow to reduce the set of possible solutions. Another observation showed that Eclipse refactoring quite often produces incompilable code (those benchmarks are marked by "$*$"), so it is likely that their implementation does not take all necessary constraints into account. For that benchmarks all errors were fixed manually prior to measuring the quality of the results.

The comparison of results in the first two columns shows that our algorithm outperforms Eclipse implementation for JavaCUP removing by 80 percent more casts, while Eclipse performs better for ANTLR. The latter result is caused by an iterator variable used to iterate over two differently typed collections. While context-sensitive analysis can detect that the variable is pointing to different collections at different places, our inference algorithm deduces raw type for iterator which causes the both collections to remain raw as well. If we split the iterator variable into two, our implementation produces more generic items than Eclipse. Since the code without such variable reuse is also easier to read, we assume the user to eliminate such reuse either by hand, or with the use of IDE provided inspection before running "Generify" refactoring.

Benchmark	LOC	Generic Items Introduced		Casts Removed		Refactoring Duration	
		Intellij IDEA	Eclipse	Intellij IDEA	Eclipse	Intellij IDEA	Eclipse
*JavaCUP v10k[6]	11048	77	84	461	257	0:40	0:7
*ANTLR 2.7.5[7]	54993	122	172	73	112	1:30	0:10
htmlparser 1.5[8]	40722	204	185	72	61	0:36	0:15
JUnit 3.8.1[9]	5217	80	82	21	21	0:7	0:7
*telnetd 2.0[10]	6866	44	43	33	33	0:5	0:2
*JLex[11]	7841	50	45	56	50	0:8	0:5
*hibernate 2.1[12]	88184	1160	1192	157	157	4:35	0:56

Fig. 1. Evaluation Results

7 Conclusion and Future Work

The solution to generification problem based on global type inference is presented. This approach is quite simple and easy to implement, while it naturally leads to obtaining type-safe solution. The number of generified items is measured to usually exceed the numbers previously reported. Moreover, in spite of the developed algorithm being exponential, our experiments show, that its running time is acceptable for use as a refactoring.

As a possible direction of the future work, we are planning to reformulate the constraint system, so that casting a raw item does not impose subtype, but rather a weaker "type is convertible" [5] constraint. To deal with context-insensitive nature of our system, another interesting experiment would be to formulate the constraint system for the SSA form [10] of the program.

References

1. Gilad Bracha, Martin Odersky, David Stoutamire, Philip Wadler. Making the future safe for the past: adding genericity to the Java programing language. ACM SIGPLAN Notices, Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vol. 33, Issue 10, 1998.
2. Daniel von Dincklage, Amer Diwan. Converting Java Classes to Use Generics. ACM SIGPLAN Notices, Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, Vol. 39, Issue 10, 2004.
3. Alan Donovan, Adam Kieżun, Matthew S. Tschantz, Michael D. Ernst. Converting Java Programs to Use Generic Libraries. ACM SIGPLAN Notices, Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, Vol. 39, Issue 10, 2004.
4. Dominic Duggan. Modular Type-Based Reverse Engineering of Parameterized Types in Java Code. ACM SIGPLAN Notices, Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Vol. 34, Issue 10, 1999.
5. Java language specification, 3rd edition. Available at http://java.sun.com/docs/books/jls/download/langspec-3.0.pdf
6. Kresten Krab Thorup. Genericity in Java with virtual types. European Conference on Object-Oriented Programming. LNCS 1241, Springer-Verlag,1997.
7. Martin Odersky, Philip Wadler. Pizza into Java: translating theory into practice. Symposium on Principles of Programming Languages. ACM, 1997.

[4] http://www.antlr.org
[5] http://htmlparser.sourceforge.net
[6] http://www.cs.princeton.edu/ãppel/modern/java/CUP
[7] http://www.cs.princeton.edu/ãppel/modern/java/JLex
[8] http://www.junit.org
[9] http://telnetd.sourceforge.net
[10] http://www.hibernate.org

8. Frank Tip, Robert Fuhrer, Julian Dolby, Adam Kieżun. Refactoring Techniques for Migrating Applications to Generic Java Container Classes. IBM Research Report RC 23238, (Yorktown Heights, NY, USA). June 2, 2004.
9. Eclipse project. http://www.eclipse.org
10. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 13 Issue 4, October 1991.