Inferred Type Instantiation for GJ

Martin Odersky

Note sent to the types mailing list, January 15th, 2002

1 Motivation

The task of type instantiation is to substitute types for formal type parameters in applications of polymorphic methods. Instantiating these parameters automatically considerably reduces clutter in program code that applies polymorphic functions. GJ uses type instantiation in a type system that combines Java's nominal subtyping discipline with F-bounded polymorphism.

In GJ, type parameters of polymorphic methods are always instantiated automatically; there exists not even a program construct for explicit instantiation. The instantiation uses only local knowledge about types of method arguments and its expected return type. Specifically, GJ's type instantiation technique is a variant of local type inference [PT98].

Pervasive inferred type instantiation as it is used in GJ places high demands on the accuracy of the inference scheme. Particularly difficult to deal with are polymorphic functions without any arguments, or polymorphic functions with type parameters that appear only in the function's result type. An example of the former is the function `Nil` declared as

```
<A> List<A> Nil() = new List<A>();
```

Given a definition such as

```
Object xs = Cons("abc", Nil()) ,
```

one would expect `Nil()` to be instantiated to type `List<String>`. Local type inference instead produces a type `List<Bottom>` for `Nil` with `Bottom` being a subtype of any reference type. Because `List` is a non-variant type constructor in GJ, `List<Bottom>` is not a subtype of `List<String>` so that the above example is rejected with a type error. This is unsatisfactory, in particular since examples like the one above are quite common.

The version of GJ distributed as the early access release for JSR14 uses instead a type instantiation scheme which relies on a special "undefined" type "`*`". That scheme would assign to `Nil()` the type `List<*>`, which is taken to be compatible with any other instance type of `List`. Alan Jeffrey [?] has recently shown that GJ's current scheme is unsound. Specifically, code that uses polymorphic methods but no explicit type casts can still cause a `ClassCastException` to be thrown at run-time. The unsoundness is caused by the semantically unclear status of the "undefined" type "*".

This note puts forward another system for inferred type instantiation for GJ which does not need to make reference to "*"-types and thus avoids the unsoundness problem of GJ's current scheme. The new scheme has been implemented in the latest version of javac. It was tested with several large examples, including the compiler itself and the GJL, a generified version of ObjectSpace's JGL. In all this code (roughly 30000 lines) there was not a single instance where a type annotation had to be given which was not there before. This is a good indication that the new type inference scheme is in practice at least as good as the previous one.

2 Informal Presentation

This section gives an informal and operational sketch of the instantiation system. Type parameter instantiation usually infers type parameters for a polymorphic method from the types of the actual value parameters of the call. For instance, assume we have a class of lists declared as follows.

```
class List<A> {
  A head;
  List<A> tail;
  List<A> prepend(A x) {
    List<A> r = new List<A>(); r.head = x; r.tail = this; return r;
  }
}
```

Then, given the declarations

```
<A> List<A> sort(List<A> xs)
List<String> xs ,
```

in the call `sort(xs)` the type parameter `A` of `sort` would be instantiated to `String`. This happens to be the only instantiation which leads to a legal typing. The following example shows that sometimes there is more than one legal instantiation:

```
<A> List<A> singleList(A x) { return new List<A>().prepend(x); }
return singleList(new Integer(1)); // returns List<Integer>
```

Here, we instantiate `A` to `Integer`, even though `Number` or `Object` would also lead to legal typings. Generally, we always instantiate type parameters to the least possible type wrt the subtype ordering which still leads to a legal typing of the application. Here's another example:

```
<A> List<A> pairList(A x, A y) { return new List<A>().prepend(y).prepend(x); }
return pairList(new Integer(1), "abc");  // returns List<Object>
```

Here, we instantiate `A` to `Object`, the least supertype of the two actual argument types `Integer` and `String`. The example shows that the instance type need not be part of any single argument type.

The "least types" strategy is a pragmatic choice, which in most cases tends to correspond to the user's intentions. Note that this choice does sometimes lead to type errors later on. For instance, if we write

```
List<Number> xs = singleList(new Integer(1));   // type error
```

we get a type mismatch, since `List<Integer>` does not conform to `List<Number>`. Nevertheless, these cases are in practice relatively rare, and they can always be fixed by adding an explicit widening cast. For the previous example:

```
List<Number> xs = singleList((Number) new Integer(1));
```

The scheme described so far works reasonably well, but it breaks down when the actual parameters contain no type information that can be used for instantiation. For instance, what type should we assign to `singleList(null)`? The actual parameter `null` is compatible with every reference type, and therefore gives no clue what the instance type should be. A more common case is when the type parameter of a polymorphic method does not occur in the types of its value parameters, as in:

```
<A> List<A> Nil() { return new List<A>(); }
```

Here, the call `Nil()` itself contains no information how the type parameter `A` of `Nil` should be instantiated.

The central idea of the new type inference scheme is to assign a *type scheme* to undetermined calls such as `Nil`. Type schemes differ from types in that they contain type variables which need to be instantiated in the context where the call appears. To distinguish such “instantiatable type variables” from the fixed type parameters of enclosing classes and methods, we append their names with a question mark. E.g. the type scheme of `Nil()` would be `List<A?>`. In the formal type system, instantiatable type variables are instead bound by a “forall” quantifier.

Type variables in type schemes are instantiated from information that’s known from the context. For example, consider the assignment

```
List<String> xs = Nil();  // instantiate A? to String
```

Here, the type variable `A?` in the type scheme `List<A?>` of `Nil` is instantiated to `String` because this is the only choice that makes the assignment well-typed. As in the case of method calls, sometimes there is more than one choice, however. An example is:

```
Object ys = Nil(); // instantiate A? to Object
```

In this case, we instantiate type variables to the largest possible type wrt the subtype ordering which makes the assignment legal. Note that we could not have chosen the least type instead, because that type does not in general exist.

Contexts such as method return expressions, where assignment compatibility to a known type is required, are treated analogously to assignments. Contexts such as casts, where nothing is known about the required type of an expression, are treated like an assignment to `Object`. Things become more complicated if the context of a type scheme is again a method call. Here’s an example:

```
<B> List<B> cons(B x, List<B> xs) { return xs.prepend(x); }
return cons("abc", Nil());    // returns List<String>
```

This example is dealt with as follows. The context of the type scheme `List<A?>` of `Nil()` is the formal parameter type `List<B>`. Hence we treat the instantiation as in the assignment

```
List<B> xs = Nil()   /// instantiate A? to B
```

That is, we instantiate `A?` to `B`. Once this is done, we instantiate the parameter `B` of `cons` as usual as the least solution of the constraints

```
String <= B, List<B> <= List<B> .
```

This gives `String` as an instance type for `B`, which is what we want.

In the last example we were “lucky” because the formal parameter type matched the actual parameter type scheme for any instantiation of the type parameter `B`. There are other situations where luck runs out, as in:

```
<B> List<B> singleList(B x) { return new List<B>().prepend(x); }
return singleList(Nil());   // returns List<List<Object>>
```

Here, no instantiation of `List<A?>` matches the formal parameter type `B`. In this case, we follow a backup strategy by replacing the type variables of the other procedure by “unknown types” (here written “`?`”) and trying the instantiation of the type scheme again. This instantiation proceeds as before, but with the added convention that an equality or subtype constraint with `?` on its right-hand side always succeeds. In other words, we instantiate relative to constraints that contain partial type information, where `?` denotes missing information. Similar constraints are used in colored local type inference [OZZ01]. In our example, since every instantiation of `List<A?>` is a subtype of `?`, we pick the largest one, i.e. `List<Object>`.

The introduction of the unknown type `?` might look perilously close to the undefined type `*` of the previous, unsound type parameter instantiation scheme of GJ. But the two play completely different roles: We use the `?` type only as an operational device in patterns which guide instantiatiation of type schemes. It does not form part of instantiated types, and hence has no effects on the run-time behavior of programs. In fact, the formal treatment of the next section makes no reference to unknown types at all; it uses type variables instead.

An alternative, slightly simpler method would instantiate a type scheme which does not match its formal parameter type to the maximal possible type without further looking at the structure of the formal type. This scheme does not require the propagation of partial type information. In the previous example, both schemes lead to the same result. Here’s an example where they lead to different results.

```
interface Function<A,B> {
  B apply(A x);
}
class Id<A> extends Function<A,A> {
  public A apply(A x) { return x; }
}
<A> Id<A> identity() { return new Id<A>(); }
<B> B applyToString(Function<String,B> f) { return f.apply("abc"); }
return applyToString(identity());
```

The application `identity()` has the type scheme `Id<A?>` which conforms to `Function<A?,A?>`. That type scheme does not match `apply`’s formal parameter type `Function<String,B>`. Replacing `B` by an unknown type yields the constraint

`Id<A?> <: Function<String,?>`	which is equivalent to
`Function<A?,A?> <: Function<String,?>`	which is equivalent to
`A? = String` .	

Hence, `A?` gets instantiated to `String`, as expected. On the other hand, if we had instantiated `Id<A?>` without regard to the partial formal parameter type, we would have ended up with `Id<Object>`. This type is not a subtype of `Function<String,B>` for any type `B`, and therefore a type error would result.

Other Aspects

The description so far covers the main principles of the new type parameter instantiation scheme. Two aspects have not been covered yet: bounds and method member selection. Here are quick sketches how these are handled.

Bounds. Type parameter bounds are taken into account as follows. When instantiating a type variable of a type scheme to its maximal type, the bound is taken into account. That is, we instantiate the type variable to a type which is a subtype of both the type given by the context and the variable’s bound type. When picking the least solution of a method’s type parameter, bounds are checked in a second stage. That is, we first minimize the type parameter and then check that it conforms to its bound.

Method Selection. Type scheme instantiation should be delayed when a type scheme is assigned to an expression from which a method member is then selected. Here's an example:

```
List<String> xs = Nil().prepend("abc")
```

If we instantiate immediately the type scheme `List<A?>` of `Nil()`, we would get `List<Object>`. Hence, the right hand side of the assignment would also yield `List<Object>` and the variable definition would be ill-typed. We avoid this problem by propagating the type variable `A?` outwards, so that it becomes an additional type parameter of the method call.

In our example, the method call `prepend("abc")` is typed as if `prepend` had the type signature `<A> List<A> prepend(A x)`. This lets `A` be instantiated to `String`, which is what we want.

3 Type System

We now make precise the intuitions developed in the previous section by means of a formal type system that handles type parameter instantiation. To simplify the presentation we omit bounds and method selection.

Setting

We consider the following small language:

Type variables	α, β, γ		
Classnames	c, d, e		
Classtypes	C, D, E	$::=$	$c\langle\overline{T}\rangle \mid \top$
Types	R, S, T	$::=$	$C \mid \overline{T} \rightarrow T$
Type schemes	U	$::=$	$\forall\alpha.U \mid T$
Variables	x, y, z		
Expressions	e, f	$::=$	$x \mid e\ e \mid \Lambda\alpha.e \mid e\langle T\rangle$
Environments	Γ, Σ	$::=$	$\epsilon \mid \Gamma, c\langle\overline{\alpha}\rangle \leq \overline{C} \mid \Gamma, \alpha \mid \Gamma, x : U$

Here, we use $\overline{X}$ for a list $X_1, \ldots, X_n$ of X elements, where the length n of the list is arbitrary. The comma operator on environments is taken to be associative and commutative. We assume that the same name is not bound twice in an environment. Therefore, environments can also be seen as sets of bindings. We implicitly assume that all names occurring in types of variables or bounds of class names are also bound in the environment.

Abbreviations for $n \geq 0$:

$$\begin{array}{lcl} \forall\alpha_1, \ldots, \alpha_n.T & = & \forall\alpha_1 \ldots \forall\alpha_n.T \\ \Lambda\alpha_1, \ldots, \alpha_n.T & = & \Lambda\alpha_1 \ldots \Lambda\alpha_n.T \\ e\langle T_1, \ldots, T_n\rangle & = & e\langle T_1\rangle \ldots \langle T_n\rangle. \end{array}$$

Subtyping

Subtyping judgements $\Gamma \vdash S \leq T$ are derived as usual.

(REFL-$\leq$) $\quad \Gamma \vdash T \leq T$

$$(\textsc{Trans-}\leq)\quad \frac{\Gamma \vdash R \leq S \qquad \Gamma \vdash S \leq T}{\Gamma \vdash R \leq T}$$

$$(\textsc{Top-}\leq)\quad \Gamma \vdash T \leq \top$$

$$(\textsc{Taut-}\leq)\quad \frac{i \in \{1, \ldots, n\}}{\Gamma, c\langle\overline{\alpha}\rangle \leq D_1, \ldots, D_n \vdash c\langle\overline{T}\rangle \leq [\overline{T}/\overline{\alpha}]D_i}$$

$$(\textsc{Arrow-}\leq)\quad \frac{\Gamma \vdash T' \leq T \qquad \Gamma \vdash \overline{S} \leq \overline{S}'}{\Gamma \vdash \overline{S} \to T \leq \overline{S}' \to T'}$$

Environment Ordering

The relation $(\preceq)$ on environments is defined to be the smallest partial order that satisfies

$$\Gamma \quad \preceq \quad \Gamma, c\langle\overline{\alpha}\rangle \leq \overline{D}$$

for all environments Γ, class names c, type variables $\overline{\alpha}$, and class types $\overline{D}$.

$(\preceq)$ expresses an "open world assumption", where the global environment may be enriched my further classes, but existing classes cannot be modified.

As an alternative, one could also use a stronger ordering $(\preceq')$ which satisfies

$$\begin{array}{lcl} \Gamma & \preceq' & \Gamma, c\langle\overline{\alpha}\rangle \leq \overline{D} \\ \Gamma, c\langle\overline{\alpha}\rangle \leq \overline{D} & \preceq' & \Gamma, c\langle\overline{\alpha}\rangle \leq \overline{D}, E \end{array}$$

for all environments Γ, class names c, type variables $\overline{\alpha}$, and class types $\overline{D}, E$. This represents a stronger open-world assumption which allows the addition of further base classes to classes that already exist. In the remainder, we will use $(\preceq)$, but the technique applies equivalently to $(\preceq')$.

Solutions

Consider subtyping judgements $\Gamma \vdash \overline{T} \leq \overline{T}'$ and let $\overline{\alpha}$ be a list of type variables disjoint from $dom(\Gamma)$. An $\overline{\alpha}$ *solution* of $\Gamma \vdash \overline{T} \leq \overline{T}'$ is a substitution σ with $dom(\sigma) \subseteq \{\overline{\alpha}\}$ and $\Gamma \vdash \sigma\,\overline{T} \leq \sigma\,\overline{T}'$. (A *substitution* is an idempotent mapping from type variables to types. The domain $dom(\sigma)$ of a substitution σ is the set of variables which are not mapped to themselves by σ.)

We introduce the following ordering on $\overline{\alpha}$ solutions σ, σ': $\sigma \leq \sigma'$ if there is a substitution ρ with $dom(\rho) \subseteq \{\overline{\alpha}\}$ such that $\rho\,(\sigma\,\alpha) \leq \sigma'\,\alpha$, for all α.

An $\overline{\alpha}$ solution σ of $\Gamma \vdash \overline{T} \leq \overline{T}'$ is *maximal* (respectively, *minimal*), if for all environments $\Sigma \preceq \Gamma$ and $\overline{\alpha}$ solutions ρ of $\Sigma \vdash \overline{T} \leq \overline{T}'$ one has $\rho \leq \sigma$ (respectively, $\sigma \leq \rho$.)

Instantiation

The judgement

$$\{\overline{\alpha}\}, R\,;\Gamma \vdash e : U \;\textbf{inst}\; e' : T$$

states that an expression e of type scheme U is instantiated to an expression e' of type T under the environment Γ and required type R. $\{\overline{\alpha}\}$ is a set of type variables in R that are "don't cares", i.e. occurrences of these variables may be replaced by arbitrary types. It is not even required that different occurrences in R of the same type variable from $\{\overline{\alpha}\}$ be replaced by the same type.

First, an auxiliary definition:

Definition. Two types S and T are $\overline{\alpha},\overline{\beta}$-similar, written $S \approx_{\overline{\alpha},\overline{\beta}} T$, if $\forall\gamma.\ [\gamma/\overline{\alpha}]S = [\gamma/\overline{\beta}]T$.

Now, for the typing rules for instantiaton:

(Inst-refl) $\emptyset\,;R\,;\Gamma \vdash e : T \ \mathbf{inst}\ e : T$

(Inst-$\forall$) $$\frac{\sigma \text{ max } \overline{\alpha} \text{ solution of } \Gamma \vdash T \leq R}{\emptyset\,;R\,;\Gamma \vdash e : \forall\overline{\alpha}.T \ \mathbf{inst}\ e\langle\sigma\overline{\alpha}\rangle : \sigma T}$$

(Inst$_1$) $$\frac{\emptyset\,;R\,;\Gamma \vdash e : U \ \mathbf{inst}\ e' : T}{\{\overline{\alpha}\}\,;R\,;\Gamma \vdash e : U \ \mathbf{inst}\ e' : T}$$

(Inst$_2$) $$\frac{R \approx_{\overline{\alpha},\overline{\beta}} R' \qquad \emptyset\,;[\overline{S}/\overline{\beta}]R'\,;\Gamma \vdash e : U \ \mathbf{inst}\ e' : T}{\{\overline{\alpha}\}\,;R\,;\Gamma \vdash e : U \ \mathbf{inst}\ e' : T}$$

The first two typing rules apply if the required type R is completely defined. In this case, monotypes instantiate to themselves (Inst-Refl), and bound type variables in type schemes are instantiated to maximal types such that the resulting instantiated type is still a subtype of the required type.

The remaining two typing rules apply if the required type is partially defined, i.e. if it contains "don't care" variables $\overline{\alpha}$. In this case the type is either instantiated without replacing type variables $\overline{\alpha}$ in the required type (Inst$_1$), or it is instantiated after replacing every occurrence of a "don't care" variable in the required type with an arbitrary type.

To ensure coherence, rules (Inst$_1$) and (Inst$_2$) need to be prioritized. Rule (Inst$_2$) is allowed to be applied to some premises in a proof tree only if rule (Inst$_1$) is not applicable to the same premises. Equivalently, this could be expressed by taking the meta-condition

$$\neg\exists e', T'.\ \emptyset\,;R\,;\Gamma \vdash e : U \ \mathbf{inst}\ e' : T'$$

as additional premise to rule (Inst$_2$).

Type Assignment

Judgements for type assignment have the form $\Gamma \vdash e : U \hookrightarrow e'$. This states that under environment Γ the expression e has type scheme U and translates to expression e'. Rules for deriving these judgements are as follows.

(Taut) $\Gamma, x : U \vdash x : U \hookrightarrow x$

(TLam) $$\frac{\Gamma, \alpha \vdash e : U \hookrightarrow e'}{\Gamma \vdash \Lambda\alpha.e : \forall\alpha.U \hookrightarrow \Lambda\alpha.e'}$$

(Tapp) $$\frac{\Gamma \vdash e : \forall\alpha.U \hookrightarrow e'}{\Gamma \vdash e\langle T\rangle : [T/\alpha]U \hookrightarrow e'\langle T\rangle}$$

$$(\textsc{App}) \quad \frac{\begin{array}{c}\Gamma \vdash f : \forall\overline{\alpha}.(R_1, \ldots, R_n) \to R' \hookrightarrow f' \\ \Gamma \vdash e_i : U_i \hookrightarrow e_i' \qquad \{\overline{\alpha}\} \,;\, R_i \,;\, \Gamma \vdash e_i' : U_i \textbf{ inst } e_i'' : T_i \qquad (i = 1, \ldots, n) \\ \sigma \text{ min } \overline{\alpha} \text{ solution of } \Gamma \vdash T_1 \leq R_1, \ldots, T_n \leq R_n \qquad \overline{\alpha}' = \overline{\alpha} \setminus dom(\sigma)\end{array}}{\Gamma \vdash f(e_1, \ldots, e_n) : \forall\overline{\alpha}'.\sigma R' \hookrightarrow \Lambda\overline{\alpha}'.f'\langle\sigma\overline{\alpha}\rangle(e_1'', \ldots, e_n'')}$$

The only non-standard rule is (App), which assigns a type to the application of a function f to a list arguments $e_1, \ldots, e_n$. Function and arguments can be polymorphic. Say, f has type scheme $\forall\overline{\alpha}.(R_1, \ldots, R_n) \to R'$ and each argument e_i has type scheme U_i. As a first step, each U_i is instantiated with f's corresponding formal parameter type R_i as the required type, and f's type variables $\overline{\alpha}$ as "don't cares". Then, a minimal $\overline{\alpha}$ solution is determined which makes the instantiated argument types subtypes of the formal parameter types. The solution might map some type variables in $\overline{\alpha}$ to themselves which indicates that these type variables are not constrained from below. These type variables are represented by the list $\overline{\alpha}'$ in rule (App). Unconstrained type variables are further propagated as universally quantified type variables in the application's result type.

The follwing proposition, while plausible, still has to be proved in detail:

Proposition 3.1 (Soundness) If $\Gamma \vdash e : \forall\overline{\alpha}.T \hookrightarrow e'$ then there exist types $\overline{S}$ such that $\Gamma \vdash e' : [\overline{S}/\overline{\alpha}]T$.

The proposition implies soundness of the inference scheme, in the sense that type parameter instantiation preserves the types of expressions.

Corollary 3.2 (Type Preservation) If $\Gamma \vdash e : \forall\overline{\alpha}. \hookrightarrow e'$ then $\Gamma \vdash e' : T$.

4 Discussion

This note has presented a new scheme for type parameter inference in GJ. We have given both an operational description and a formal type system. The scheme has been implemented in the present version of javac. It seems to work well in practice, and its soundness should be easy to establish.

Admittedly the details of the type inference scheme are somewhat intricate, from an operational as well as from a formal viewpoint. It seems to be hard to come up with an alternative that is both easy to specify and convenient to use for programmers. A range of alternatives has been considered and rejected. Here are just three examples.

1. No Type Parameter Instantiation

One could require type parameters to be always given explicitly. In practice, this makes program using generic libraries very unwieldy. Users would be tempted to do without generics altogether in order to avoid the notational overhead.

2. Only Infer Type Parameters from Arguments

One could require explicit type parameters whenever type parameters cannot be inferred from function arguments. However, this also clutters programs to some degree. Furthermore, users are likely to forget type parameters frequently, because they are not needed for every call of a polymorphic method. This leads to frequent annoying compiler error messages. For instance, the following would be in error:

```
List<String> xs = Nil();
```

One would need to write

```
List<String> xs = Nil<String>();
```

Another objection to explicit type parameters is that there is no good syntax in Java to express them. The code above uses

```
method-name < type-parameters > ( value parameters ) .
```

However, this leads to an ambiguous grammar: The string

```
a ( b < c, d > ( e ) )
```

can be parsed as a function call applied to two boolean arguments:

```
a( (b < c), (d > (e)) )
```

Or it can be parsed as a two nested function call, where the inner call takes type parameters:

```
a( b <c,d> (e) )
```

3. Use Global Constraint Solving

Instead of making type parameter inference less powerful, one might also consider going the other way. The proposed type inference scheme is *local*, in the sense that constraints are derived from a single construct and then immediately a solution to these constraints is determined. One might also consider collecting constraints for whole expressions, methods, or even programs, which are then solved globally. This is what is done conceptually in Hindley/Milner style type inference, which is used in languages such as ML, Haskell, or Mercury. Hindley/Milner type inference reduces a typing problem to a set of equality constraints between types, which are then solved using unification.

Since GJ has subtyping, one would need to use subtyping constraints instead. Unfortunately, object-oriented languages with nominal type systems such as Java are incompatible with the global constraint solving approach. To see one reason why (there are several others), consider the following two simple subtyping constraints, where `A?` and `B?` denote instantiatable type variables.

```
List<A?> <: B?
List<Number> <: B?
```

Two solutions to these constraints are:

```
A? = Number, B? = List<Number>
A? = Integer, B? Object
```

No solution is better (wrt the subtyping ordering) than the other, and there is no third solution that improves on both.

The example shows that any constraint system which admits instantiatable type variables on both sides of a subtyping constraint does not have a single best solution. Hence, any type system that relies on solutions of such constraints does not have principal types. Therefore, our type parameter instantiation scheme was carefully designed to avoid such constraints.