

# Proving Type Soundness: Featherweight GJ

Eric Allen, Rice University, eallen@cs.rice.edu

19 June 2001

## Abstract

I present a review of Featherweight GJ, a small language containing many of the features of Java, but with support for generic types. I discuss the steps involved in proving the type-soundness of this language. I then discuss what parts of this proof could be reused for an extension of Featherweight GJ that supports run-time operations on generic types.

## 1 Introduction

A type soundness proof for a language provides us with a guarantee that the invariants of the type system will not be violated at runtime. Formally, a type soundness claim is of the following form:

If a program expression  $e$  has type  $T$ , then one of the following conditions hold:

1. The expression evaluates to a value of type  $S$ , where  $S \leq T$ .
2. Expression evaluation never terminates.
3. Expression evaluation results in one of a predefined set of errors (e.g., division by zero, array out-of-bounds, casting failure, etc.).

Since Wright and Felleison's "A Syntactical Approach to Type Soundness", type soundness proofs for programming languages have been carried out in the context of an operational semantics, and have generally consisted of the following two steps:

1. *Subject Reduction*: The type of an expression is preserved at each step in its evaluation.
2. *Progress*: Evaluation of a well-typed expression never gets "stuck". For example, it must be shown that data structures actually contain the fields accessed from them, method calls succeed, etc.

Typically, the strategy used for proving subject reduction is induction over the one-step evaluation of  $e$  to expression  $e'$ . Progress lemmas are generally proven by considering the constraints ensured by the semantics before the various reduction rules can be applied.

## 2 Featherweight GJ

Featherweight GJ is a toy version of the GJ language, which is itself an extension of Java with generic types. The syntax, typing rules, and evaluation rules for Featherweight GJ are included on the following pages. A program in Featherweight GJ consists of a set of class definitions, and an expression (equivalent to the *main* method of the program). Notice the following features of this language:

- There is no way to modify the value of any variables.
- There is no null value or corresponding type.
- There are no static fields.
- There are no conditional statements or expressions.
- There are no interfaces.
- There is no *instanceof* operator.
- *New* expressions and casts cannot be applied to type variables.
- The “base types” of GJ do not exist in this language.
- There are restrictions on the downcasting of parametric types (discussed below).

Despite these restrictions, Featherweight GJ models several of the most salient properties of GJ. Namely,

- The bounds of type variables must be non-variable types. This prevents the necessity of using a fixed-point semantics to determine type variable bounds.
- Type variables are in the scope of other bounds, as well as the extends clause.

The semantics of computation in Featherweight GJ consist of three basic rewrite rules for the operations of field access, method invocation, and casting. Additionally, there are “congruence” rules to allow for the evaluation of an expression through the application of the three basic rules to its sub-expressions.

To see how these evaluation rules are applied to a program, let’s consider the following set of class definitions:

```
class List<T extends Object> extends Object { List(){} }
class Empty<T extends Object> extends Object {
  Empty() { super(); }
}
class Cons<T extends Object> extends List<T> {
```

```

    T first;
    List<T> rest;
    Cons(T first, List<T> rest) {
        super();
        this.first = first;
        this.rest = rest;
    }
    Cons<T> spawn(T newFirst) {
        return new Cons<T>(newFirst, this.rest);
    }
}

```

Now consider the following expression in the context of these class definitions:

```
new Cons<Object>(new Object(), new Empty<Object>())
```

Since this expression contains no field accesses, method invocations, or casts, none of the reduction rules apply to it, and it is therefore in simplest form. In fact, the set of values of Featherweight GJ consists of all *new* expressions that themselves contain only *new* expressions.

Let's consider another expression in this context:

```
new Cons<List<Object>>(new Empty<Object>(), new Empty<List<Object>>()).
  spawn(new Cons<Object>(new Object(), new Empty<Object>()))
```

By application of rule GR-Invk, we can reduce this to:

```
new Cons<List<Object>>(new Cons<Object>(new Object(), new Empty<Object>()),
  new Cons<List<Object>>(new Empty<Object>(),
    new Empty<List<Object>>()).rest)
```

And, by application of GRC-NewArg (via GR-Field):

```
new Cons<List<Object>>(new Cons<Object>(new Object(), new Empty<Object>()),
  new Empty<List<Object>>())
```

This last expression is in simplest form.

### 3 Typing Featherweight GJ

Unlike many languages, the typing of a program in Featherweight GJ requires two environments:

1. A type environment  $\Gamma$  mapping terms to their types.

2. A type-bound environment  $\Delta$  mapping types to their upper bounds.

This is necessary because of the mixing of subtype polymorphism and parametric polymorphism in the Featherweight GJ type system.

As one might expect, a type-bound environment  $\Delta$  maps types to their upper bounds, according to the following rules:

1. The bound of a type variable is the non-variable type it was declared to be in the scope of an expression.
2. The bound of a non-variable type is itself.

### 3.1 Well-formed Types

In the context of a given type-bound environment  $\Delta$ , there are restrictions as to what constitutes a well-formed type. The rules are as follows:

1. *Object* is a well-formed type in any type-bound environment.
2. If type variable  $X$  is in the domain of  $\Delta$  then  $X$  is well-formed in  $\Delta$ .
3. If class  $C$  has type parameters  $\overline{X}$  with upper bounds  $\overline{N}$  then any instantiation of  $C$  with types  $\overline{T}$  s.t.  $\Delta \vdash \overline{T} \leq [\overline{T}/\overline{X}]\overline{N}$  is a well-formed type in  $\Delta$ .

### 3.2 Subtyping

Similarly, the rules for the subtype relation in the context of a type-bound environment are as follows:

1. The subtype relation is reflexive.
2. The subtype relation is transitive.
3. A type variable is a subtype of its bound in  $\Delta$ .
4. If class  $C$  has type parameters  $\overline{X}$  with upper bounds  $\overline{N}$  then any well-formed instantiation of  $C$  with types  $\overline{T}$  is a subtype of  $[\overline{T}/\overline{X}]\overline{N}$ .

### 3.3 Valid Downcasts

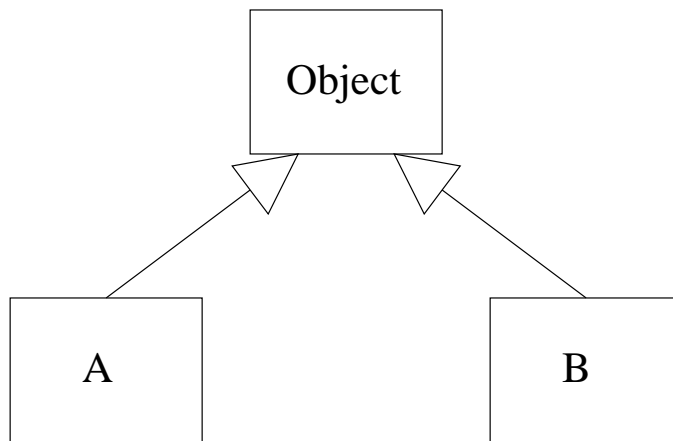
As mentioned above, there are restrictions on which types an expression can be downcast to. In particular, an expression  $e$  of type  $T$  may be downcast only to another type  $S$  with an equal number of type parameters. This restriction is needed so that the commutativity result of Featherweight GJ holds. This commutativity result relates Featherweight GJ's "type-carrying" semantics to a semantics based on type-erasure. Intuitively, one can see that, without the restriction on downcasts, the set of casts that succeed would be different in these two semantics. Consider, for example, the following expression:

`(List<D>)o`

Where `o` has static type *Object*, but is a *List<C>* at runtime (where  $C \neq D$ ). Without the above restriction, the cast would succeed under the type-carrying semantics, but not under the semantics based on type-erasure.

### 3.4 Stupid Casts

Stupid casts are casts in which the static type of an expression and the type it is being cast to are unrelated. For example, consider the following class hierarchy:



Clearly, it would be pointless to cast an expression of type *B* to type *A*. In Featherweight GJ, such “stupid casts” are not allowed in a program before any evaluation rules have been applied. However, it is necessary to include a typing rule for stupid casts because they can arise while evaluating legal expressions. Since type soundness for Featherweight GJ is proven by means of subject reduction, expressions containing stupid casts must be typed in order for subject reduction to hold. For example, consider the following expression:

`(A)(Object)new B()`

Both casts in this expression are valid: the first is an upcast from type *B* to type *Object*, and the second is a downcast from type *Object* to type *A*. But the expression can be reduced through evaluation to:

`(A)new B()`

which contains a stupid cast. But if subject reduction is to hold, the type of this second expression must be a subtype of *A*. Nevertheless, the issue of stupid casts was overlooked in Flatt, Krishnamurthi, and Felleisen’s published type soundness proof of Classic Java.

## 4 The Type Soundness of Featherweight GJ

We can now formally state a subject reduction theorem for Featherweight GJ:

If  $\Delta \Gamma \vdash e \in T$  and  $e \mapsto e'$  then  $\Delta \Gamma \vdash e' \in T'$  for some  $T'$  s.t.  
 $\Delta \vdash T \leq T'$ .

Similarly, we can state a progress theorem for Featherweight GJ:

Suppose  $e$  is well-typed. Then the following conditions hold:

1. If  $e$  includes an expression of the form  $new N_o(\bar{e}).f$  as a subexpression, then  $fields(N_o) = \overline{T} \overline{f}$  and  $f \in \overline{f}$ .
2. If  $e$  includes an expression of the form  $new N_o(\bar{e}).m \langle \overline{V} \rangle (d)$  as a subexpression, then  $mbody(m \langle \overline{V} \rangle, N_o) = (\overline{x}, e_0)$  and  $\#(\overline{x}) = \#(\overline{d})$ .

The proof of these two theorems in the Featherweight GJ paper is preceded by twelve supporting lemmas. These lemmas prove various intuitively plausible facts such as “If a type  $S$  is a subtype of  $T$ , then the fields of the bound of  $S$  are a superset of the fields of the bound of  $T$ .” (Lemma 3.4.8). Each supporting lemma is proven by induction over the structure of the proof of the type of an expression. Of course, in general, one cannot prove a theorem of the form  $A \rightarrow B$  by inducing over the structure of the proof of  $A$  (as Godel has shown, there may not be any such proof!). The strategy works in the case of type soundness proofs because an expression  $e$  has type  $T$  iff there is a proof of it.

Subject reduction proceeds by induction over the one-step evaluation of  $e$  to  $e'$ . To see how the proof proceeds, let us consider one case in the induction: that in which the evaluation rule applied was GRC-Field. Then  $e = e_0.f_i$ ,  $e' = e'_0.f_i$ , and  $e_0 \mapsto e'_0$ . Since this expression is well-typed, it must have been typed according to the rule GT-Field. Therefore, we know that  $\Delta \Gamma \vdash e_0 \in T_0$ , for some  $T_0$  s.t.  $fields(bound_\Delta(T_0)) = \overline{T} \overline{f}$ , and  $e \in T_i$ . Then, by the induction hypothesis,  $e'_0 \in T'_0$  for some  $T'_0$  s.t.  $T'_0 \leq T_0$ . We can now apply Lemma 3.4.8, mentioned above, to infer that the fields of the bound of  $T'_0$  are a superset of the fields of the bound of  $T_0$ . Therefore, they include field  $f_i$ , with type  $T_i$ . But this means that, according to the rule GT-Field,  $e_0.f_i$  also has type  $T_i$ , so we are done.  $\square$

I will now consider how an extension of Featherweight GJ to support run-time operations on parametric types would affect the proof of these lemmas, and, therefore, a proof of the type-soundness of such an extension.

## 5 Featherweight NextGen

When considering the possible extensions of Featherweight GJ to a language supporting run-time operations on parametric types, it is important to remember that some of the fundamental limitations of Featherweight GJ prohibit extension in the same ways that NextGen extends GJ. First, recall that Featherweight GJ does not have an *instanceof* operation for use with any type.

Adding such an operation would be useless anyway, since there are no conditional statements. Additionally, classes in Featherweight GJ are restricted to a single constructor. This constructor must take a number of arguments equal to the number of fields in the class, and initialize them all. Any change to this restriction would quickly increase the complexity of the language, particularly because Featherweight GJ has no null value. Because of these restrictions on constructors, it would be difficult to allow expressions of the form  $new T(\overline{f})$ , where  $T$  is a type variable. What constructor would one call on such a type? The only constructor that the language could reasonably require on all possible instantiations would be a zeroary constructor. But this would necessarily prohibit the existence of any other, non-zeroary, constructors for these types! This is clearly unacceptable. Considering that *new* expressions of this form are of limited value anyway, I will not consider an extension of Featherweight GJ that includes such expressions.

This leaves us with only one runtime operation on type variables that can be easily added to the language and provide a significant increase in expressiveness: the casting to a type variable. Extending the language in this way would require rewriting the existing computation rules for casting, as well as the static typing of cast expressions. Additionally, the following two changes would have to be made:

1. The “stupid cast” rule would have to be split into two cases, depending on whether the cast is to a type variable.
2. The constraints on downcasting would have to be relaxed.

The first change is necessary because, when casting to a type variable, a stupid cast is not so stupid. Although it is rare that one can determine a subtype relationship between the type of an expression and a type variable, such type variables may be instantiated with types that make the cast perfectly legitimate. Therefore, if we are to allow for the casting to a type variable, it would be unreasonable to prevent the vast majority of such casts by leaving the existing stupid cast rule unchanged. Nevertheless, we should still reject programs that contain a stupid cast to a non-variable type.

The second change would directly affect the structure of the type soundness proof for Featherweight NextGen, and I will discuss it below. But first, let’s examine the issue of the increased expressiveness of the extended language. We can be sure that the expressiveness of the language is increased by considering an important commutativity result for Featherweight GJ.

## 5.1 Commutativity with Type Erasure

One important feature of Featherweight GJ is that we can prove a commutativity theorem that relates its semantics to another possible semantics for the language based on type erasure. This is useful because it helps to ensure that the implementation of GJ, which is based on the notion of type erasure, is, in fact, correct. In fact, the initial attempt to prove this theorem uncovered a

bug in an early implementation of the GJ compiler. Although the notion of type erasure can be defined formally, I will not do so here, since such a formal treatment is not necessary for the present discussion. Instead, I will rely on the intuitive notion of type erasure as discussed in the original paper on GJ.

Before presenting the final form of this commutativity theorem, I will first mention what one might naively expect it to say. Ideally, we would want the following commutativity result to hold:

The erasure of a reduction of a Featherweight GJ expression is identical to some reduction of its erasure.

Unfortunately, such a commutativity theorem is too strong for Featherweight GJ. Consider, for example, the following expression:

```
new Cons<A>(a,b).first
```

This expression reduces to

```
a
```

However, its erasure,

```
(A)new Cons(|a|,|b|).first
```

reduces to

```
(A)|a|
```

where vertical bars are used to signify the erasure of a subexpression. The added cast cannot be eliminated in the next reduction (thereby matching  $(A)|a|$  to the erasure of the reduction,  $|a|$ ) unless  $|a|$  is a *new* expression. Otherwise, we would first have to apply some reduction rule to the subexpression  $|a|$ , preventing the entire expression from reducing to  $|a|$ . Furthermore, expressions that include method invocations may, after erasure, reduce to expressions with arbitrarily many additional casts. So, the level of commutativity discussed above cannot hold for Featherweight GJ. Luckily, we know that all of the added casts will succeed (by virtue of the fact that the original expression passed type checking). Therefore, the following, modified, commutativity result does hold:

The erasure of a reduction of an expression  $e$  will, after some further reduction, produce an expression identical to some reduction of the erasure of  $e$ .

Now, such a result could not possibly hold for an extension of Featherweight NextGen, as I've described it. This is because type variables erase to the erasure of their upper bounds (or, at least they do in GJ. In Featherweight GJ, type variables are all eliminated by the erasure of parametric types, and so there is no need to erase them). Indeed, this is exactly why a simple type erasure strategy

cannot be used in an implementation of NextGen. Added type information must be kept at runtime.

Because such a result cannot hold for our extension, we know that it does indeed add expressive power to the language. I will now consider what changes to the type soundness proof of Featherweight GJ must be made to accommodate this extension.

## 5.2 Downcasts Revisited

Recall that Featherweight GJ imposed restrictions on how parametric types could be used in a downcast. Namely, an expression with type  $T$  could be downcast only to a type  $S$  with an equal number of parameters. In the extension to Featherweight NextGen described above, this restriction must be eliminated. The reason is that the set of type parameters of an instantiation of a type variable is not fixed. But this restriction is used in the proof of type soundness, so its elimination would have ramifications on how much of the type soundness proof of Featherweight GJ could be used for Featherweight NextGen. In particular, Lemma 3.4.3 relies on the existence of this restriction. That lemma can be paraphrased as follows:

If  $C$  can be downcast from  $D$ , and  $C \langle \overline{T} \rangle$  is a subtype of  $D \langle \overline{U} \rangle$ , then  $C \langle \overline{T} \rangle$  is the only instantiation of  $C$  that is a subtype of  $D \langle \overline{U} \rangle$ .

This lemma is in turn used by Lemmas 3.4.10 and 3.4.11 to ensure that subject reduction holds in the case of parametric method invocation. Its proof relies on the restriction that  $C$  can be downcast from  $D$ . But since we are eliminating the restriction on downcasts, we are of course strengthening this assertion, and we would need to check that it still held. Furthermore, tracing the use of the modified lemma through the subsequent lemmas concerning method invocation may reveal that a still stronger lemma concerning the relation of type variables and their supertypes is needed.

## 5.3 The Possibility of a Commutativity Result for Featherweight NextGen

One interesting question to ask about our extension of Featherweight GJ is whether any erasure semantics could be defined on the language to allow for some sort of commutativity result. Intuitively, we would expect such a result, since the current implementation of NextGen relies in part on type erasure (although, as I mentioned above, additional information must be kept at runtime). When considering this issue, there are two concerns that arise immediately:

1. The method of erasure applied in NextGen involves the introduction of interfaces for erased parametric types. This is unavoidable, since some form of multiple inheritance is needed to correctly carry the necessary parametric type information at runtime. Therefore, before a corresponding

notion of type erasure could be formulated for Featherweight NextGen, interfaces would have to be added to the language. Unfortunately, several of the lemmas used in the type soundness proof of Featherweight GJ rely on single inheritance in the class hierarchy, so these lemmas would have to be re-examined.

2. The set of instantiations of a parametric class that will occur at runtime cannot be determined statically. In the implementation of NextGen, we solve this problem through the use of a modified class loader that dynamically generates the non-parametric form of the instantiated classes as needed. In order to capture this part of the implementation in an erasure semantics, we would have to define a notion of the infinitely large expansion of all instantiations of a parametric class, and their corresponding non-parametric forms.

## 6 Conclusion

From this analysis of Featherweight GJ and its corresponding type soundness theorem, we can conclude that an extension of the language to one supporting runtime operations on parametric types would involve modest revisions of the existing semantics. Furthermore, it appears that such an extension could reuse large parts of the type soundness proof for Featherweight GJ. To be sure, every induction over the proof of an expression's type would have to be reconsidered, in light of the modifications to the typing rules. But the ramifications for most of these inductions would be minimal. There would be somewhat larger ramifications for the one lemma that explicitly refers to the downcasting constraints, and to the other lemmas that rely on it. Nevertheless, it should be possible to prove similar lemmas that will allow the proof to proceed.

As for the possibility of a commutativity result for Featherweight NextGen, we believe that such a theorem can be proven (because we believe our current implementation of NextGen is correct!). But it should be mentioned that such a result is not as important for NextGen as for GJ. The original semantics provided for GJ was a type erasure semantics, and GJ is constrained specifically to allow for the possibility of such a semantics. NextGen, on the other hand, does not impose these constraints, and is therefore less closely tied to the notion of type erasure (despite the fact that our current implementation makes use of it).