Type-Preserving Compilation of FGJ

Yichen Xie  Valery Trifonov
Department of Computer Science, Yale University
P. O. Box 208285, New Haven, CT 06520-8285, USA

Abstract

Parametric polymorphism has been one of the most requested language features in the Java developer community. Previous work in integrating this feature into Java has paid much attention to the compatibility with existing JVM. This extra constraint leads to either inefficiencies or compromises in type safety during the translation. We present a type-preserving encoding of Featherweight Generic Java (FGJ) in an extension of FLINT—a simple, implementable typed intermediate language used in the optimizing compiler for SML/NJ. Our encoding faithfully translates type parameterization in FGJ into polymorphic constructs in the target language, thus maintaining type safety without introducing inefficiencies like code expansion or runtime overhead of unnecessary downcasts. The formal translation from FGJ allows comprehensible type-preserving proofs, and could serve as a starting point for extending the translation to the full Generic Java.

1 Introduction

JavaTM [5] is a strongly typed, object-oriented programming language that has been under the lime-light ever since its inception in 1995. It has now become the language of choice for Internet applications. Its machine-independent target architecture allowed true “write once, run everywhere” capabilities. Its powerful type system gives application developers and bytecode receivers the language support for guaranteed safety of the program code. These features have not only significantly shortened the software development cycle and improved productivity, but also enabled a whole new range of mobile applications that require proof of safety for execution.

However, one important feature that is sorely missing from Java is genericity, the power to express a generic construct regardless of some underlying parameter types. Programmers have adopted the “generic idiom” as a workaround for this problem. Any reference to a generic type is replaced by the universal parent class of all classes–Object. Upcasts and downcasts are inserted whenever an object of a generic type is passed to and returned from the package. This process is often tedious and error prone. Furthermore, since downcasts are checked at runtime, programming errors that could otherwise be detected at compile time are only signaled at runtime. Besides that, there is also performance penalties for downcasts, since they are checked at run-time.

Parametric polymorphism is a natural solution for this problem. Several proposals that add this much needed feature into Java has been proposed in the recent years. Yet almost all of them have the extra constraint that they must be compatible with existing JVM and also with legacy source and binary Java code. Since JVML as is currently defined leaves little room for expressing type parameters, previous proposals have to resort to either a homogeneous approach [2], where extra type information are erased at compile time; or a heterogeneous approach [1], where generic code serves as a template, and is expanded whenever instantiated; or a combination of the two [3].

We present a type preserving translation from featherweight generic Java, a core calculus for Generic Java, to an extension of the FLINT typed intermediate language. Our translation is based on Java to FLINT framework proposed by Leaue, Trifonov, and Shao [7]. We introduce concepts such as extenders and polymorphic tails of higher kind to capture type parameterization in Java. To simplify presentation, we only present a translation that supports self-recurisve bounds and class definition. So classes cannot depend on each other in a cyclic fashion. We also choose not to support downcast in this presentation. We believe these features are orthogonal to type parameterization and easy to adapt by following the steps outlined in [7].

The advantage of our translation is many fold. First, unlike the homogeneous approach, all extra type information the programmer supplies in the source language is preserved in this translation. Therefore, the generated target code will still be type-checkable. We also avoid efficiency problems such as code-expansion that is present in the heterogeneous approach. Each generic class instantiation will correspond to a type application, so instead of blindly expanding the generic “template” on each instantiation, it is left to the back-end to decide on the optimal choice of whether to specialize or not.

We describe syntax and semantics of our source and target languages in the next two sections. We will explain the translation in section 4. In this preliminary draft version of the paper, we will base our explanation on the assumption that readers are already familiar with previous work on this topic [7, 2, 1, 3]. We will also assume knowledge about FLINT and Featherweight Generic Java [6]. The upcoming extended version will instead be self-contained.

2 Source Language

The source language for our translation is Featherweight Generic Java (FGJ), a core calculus for Java extended with generic types. Syntax of the source language is given in figure 1. For reference, semantics of FGJ is reprinted in appendix A.

Kinds \[ \kappa ::= \Omega \mid R^k \mid \kappa \rightarrow \kappa' \mid \{ \ell : \kappa \} \mid X \mid \forall X. \kappa \]

Types \[ \tau ::= \alpha \mid \lambda \alpha : \kappa. \tau \mid \tau \rightarrow \tau' \mid \{ \ell = \tau \} \mid \tau ; \ell \mid \tau \mid \{ \tau \} \mid [\tau] \]

Terms \[ e ::= \bar{x} \mid \Lambda x : \tau. e \mid e e' \mid \{ \ell = e\} \mid e \lambda \alpha \mid e \bar{x} \mid \text{fix } [\tau] e \mid \text{fold } e \text{ as } \tau \text{ at } \tau' \mid \text{unfold } e \text{ as } \tau \text{ at } \tau' \mid \Lambda \alpha : \kappa. e \mid e \bar{\tau} \mid \alpha : \kappa \equiv \tau ; e \bar{\tau}' \mid \text{open } e \text{ as } (\alpha ; \kappa ; x : \tau) \text{ in } e' \mid \Lambda X. e \mid e [\kappa] \]

Figure 1: Syntax of the source language.

Figure 2: Syntax of the target language.

2 Source Language

The source language for our translation is Featherweight Generic Java (FGJ), a core calculus for Java extended with generic types. Syntax of the source language is given in figure 1. For reference, semantics of FGJ is reprinted in appendix A.

We abbreviate the keyword extends to the symbol \( \downarrow \). Metavariables \( \bar{X} \), \( \bar{Y} \), and \( \bar{Z} \) range over type variables; \( \bar{T} \), \( \bar{U} \), and \( \bar{V} \) range over types; and \( \bar{N} \) and \( \bar{O} \) range over non-variable types. We use \( \bar{X} \) to denote a possibly empty sequence \( x_1, \ldots, x_n \) (and similarly for \( \bar{T}, \bar{N}, \ldots \)).

A class declaration (CL) specifies names of the new class and its superclass (\( \bar{N} \)), a list of type parameters and their respective bounds (\( \bar{\kappa} \bar{\delta} \)), a sequence of field declarations (\( \bar{T} \bar{f} \)), a constructor (\( \bar{K} \)), and a sequence of method declarations (\( \bar{N} \bar{f} \)). Types of fields can make use of the type parameters in the declaration. Similarly, types used in the argument list and the return type of a method can depend on its type parameters. Constructors always take all the fields as arguments, in the correct order. There are five forms of expressions: variables, field selection, method invocation, object creation, and cast. Note that target type of a cast can only be a non-variable type.

There are no assignments, interfaces, super calls, exceptions, or access control. FGJ permits recursive class dependencies, but for brevity of presentation, we will only allow self-recursive dependency in our translation. This will allow a type parameter to have bound that depend on the parameter itself.

Typing judgements of FGJ are of form \( \Delta, \Gamma \vdash e : \bar{T} \). Note that unlike in FJ (and Java), covariant subtyping of method return type is allowed in FGJ (and GJ). The return type of a method may be a subtype of the return type of the corresponding method in the superclass, although the bound of type variables and the arguments must be identical. Please refer to the appendix for the rules, and \([6]\) for further details.

3 Target Language

The target language of our translation is the higher-order polymorphic \( \lambda \)-calculus \( F_\omega \) \([4, 8]\) extended with type tuples, existential types, row polymorphism, ordered records, sum types, iso-recursive types, term-level fix-point, and kind polymorphism. Syntax of this language can be found in figure 2. Semantics is given in appendix B.

For now, readers are referred to \([7]\) for detailed descriptions on features such as polymorphic records, row kinds, and iso-recursive types. Because our translation does not consider mutually recursive class definitions, conventional fold/unfold rules on recursive types suffice for our purposes. Our translation makes further extentions to the target language by adding kind polymorphism. Kind abstractions and applications at the type and term level are used to capture precisely the object type of a type parameter (unlike the homogeneous translation where type parameters are replaced by their loosest bound type), as will be shown in section 4.2.

4 Translation

4.1 Object Encoding

We extend the object encoding framework described in \([7]\) by adapting generic classes. In the non-generic case, an object is encoded as a record with values of all the fields plus a shared virtual table containing pointers to method code. To invoke a method, we select the corresponding method pointer from the virtual table of the object and apply it to the object itself. We then apply the result to the arguments of the method. In the generic case, fields and methods of an object can depend on some bounded type parameters. We use the extender constructs introduced in the previous section to model type parameterization in the source language. To invoke a method, we need use the TrmInst macro to pass the type parameters first, then we follow the usual procedure to complete the method invocation.

In figure 3, we introduce Rows, a type operator that takes extenders as argument and produces the rows containing fields and methods introduced between two classes in a subclass relationship. Intuitively Rows[\( C, A \)] contains the difference between \( C \) and \( A \), parameterized by extenders. Aside from the functions described in \([7]\), the Rows macro here has two more usages: it is used as extenders for non-variable types, and it is composed with extenders to produce Rows for variable types. Rows[\( \alpha : \kappa \)] is a helper macro that instantiates the generic Rows with actual extenders for both variable and non-variable types.

4.2 Extenders

We model type parameterization in FGJ by extenders at both the type level and the term level. The extenders are...
taken as extra parameters to type or term level constructors. They are used to construct types of objects whose static type contains variables. Term level extenders are of the form

\[(\mathcal{A} \kappa_i, \lambda \text{ext}_{i, \mathcal{A}, \mathcal{B}}) : ((\Omega \rightarrow \kappa_i) \rightarrow ((\Omega \rightarrow \text{ktail}[\mathcal{B}]))), i \in \{1, \ldots, n\}\]

Type level extenders are of similar shape and functionality. Variable \(i\) ranges over the number of type parameters. We use \(\Delta\) in the translation macros to keep track of all extenders (in this case \(\kappa_i\) and \(\text{ext}_{i, \mathcal{A}, \mathcal{B}}\)) and sub-typing relationships \((\Delta \vdash \mathcal{X}, \mathcal{Y})\). Each \(\kappa_i\) can be thought of as the equivalence of \(\text{ktail}[\mathcal{B}]\) for the type parameter \(\mathcal{X}\). The \(\text{ext}_{i, \mathcal{A}, \mathcal{B}}\) argument corresponds to \(\text{Rows}[\mathcal{X}, \mathcal{A}, \mathcal{B}]\).

Intuitively, the extender \(\text{ext}_{i, \mathcal{A}, \mathcal{B}}\) captures the difference between the type parameter \(\mathcal{X}\) and its bound \(\mathcal{B}\). We could have simply hidden this difference, since from inside of the class definition, an object of class \(\mathcal{X}\) is treated exactly the same as \(\mathcal{B}\). This is similar to the homogeneous approach, where type parameters are “erased” to their bounds. But from the outside the definition of the class, we have a more detailed view of type parameter \(\mathcal{X}\), because when a generic class is instantiated with some concrete type \(\mathcal{T}\) for \(\mathcal{X}\), it is expected to behave like a “specialized” version for \(\mathcal{T}\). Therefore, methods that return values of type \(\mathcal{X}\) should return value of type \(\mathcal{T}\), but not \(\mathcal{B}\), in which case a downcast is needed. Similar argument applies to the case when we pass values to method code of the class. Therefore, We choose to preserve this difference, and use extenders to construct the correct types “on the fly.”

To construct object type of \(\mathcal{X}\) under \(\Delta\), we go through the equivalence of \(\text{ObjTy}[\Delta, \mathcal{N}]\) construction for non-variable types, except that we do not construct intermediate steps since we only need the external view of \(\mathcal{X}\). We compose \(\text{Rows}[\Delta, \Delta(\mathcal{X}), \mathcal{N}]\), which denotes the difference between two non-variable types, with \(\text{ext}_{i, \Delta, \mathcal{X}}\) to obtain \(\text{Rows}[\Delta, \mathcal{X}, \mathcal{N}]\), which forms the basis for the type construction given in figure 3. Using similar techniques, we can construct object type of \(\mathcal{C}(\mathcal{T})\), where \(\mathcal{T}\) might refer to variable type \(\mathcal{X}\).

We tackle the self-recursive bound problem by using a polymorphic tail of higher kind. tail, as defined in [7], can now depend on the twin type. To allow mutually recursive class definitions and type parameter bounds, we will replace twin by a type record similar to \(\text{World}\). The kind of \(\text{World}\) will be more complicated, but the extension will otherwise be straightforward.

### 4.3 Expression Translation

We now look at type-directed translation of FGJ expressions. Figure 6 contains macros for \(\text{pack}\), \(\text{upcast}\) and five rules governing the judgement \(\text{exp} \vdash \Gamma; \epsilon = e\) for term translation. As usual, \(\Delta\) keeps track of sub-typing relationships between types, and the corresponding tail kinds and extentors. \(\Gamma\) tracks types for term variables. \(\epsilon\) is the FGJ expression, while \(e\) is the corresponding term in the target language.

The \(\text{pack}\) macro packages and folds an open-self term into a closed, complete object type. The \(\text{upcast}\) macro unfolds and repackages an object term to a term of some super-class. Intuitively, it hides more of the “tail” of the original term. The rest of the macros also follow the steps outlined in [7].

### 4.4 Class Encoding

Apart from defining types, classes in FGJ serve three other roles: they are extended, invoked as constructors, and used in dynamic casts. In our discussion, we omit the dynamic cast feature of FGJ and focus on the other two roles. In our translation, each class declaration is compiled into a module exporting a record with two elements—one to address each of these roles. The first role is addressed by the \text{dict} component of the class record. It is a polymorphic record of all method code with extensible tail. Method code is produced by the \text{meth} macro, which supports two cases. Method code is either inherited directly from the super-class, in which case the method body is just a call to the corresponding code of the super-class; or constructed anew. Method code translation macro also support type parameterization by taking extenders as arguments in the method body.

Inheritance is accomplished by instantiating \text{dict} with non-trivial tail; while \text{vtab}, the virtual table that is shared among all objects of this class, is constructed by instantiating \text{dict} with the trivial empty tail. The second role is addressed by the new component of the record. Constructor
in FGJ simply takes all fields as argument in the correct order. We translate the constructor as a function which takes the fields as curried arguments, places them into a record with the vtable, and then folds and packs to produce the object.

\[
\begin{align*}
\text{TypInst}[\Delta, \tau, X, N] & \equiv \tau[\bar{N}] \{ (\text{tail}, \text{Rows}[\Delta, N, \Delta(X)]) \} \\
\text{TypInst}[\Delta, \tau, 0, N] & \equiv \tau[\bar{N}] \text{[ktail]}(\text{Rows}[\Delta, 0, N]) \\
\text{TrmInst}[\Delta, \tau, X, N] & \equiv \tau[\bar{N}] \{ (\text{tail}, \text{Rows}[\Delta, N, \Delta(X)]) \} \\
\text{TrmInst}[\Delta, \tau, 0, N] & \equiv \tau[\bar{N}] \text{[ktail]}(\text{Rows}[\Delta, 0, N])
\end{align*}
\]

Figure 4: Macros for Generic Class Instantiation

\[
\begin{align*}
\text{Dict}[\Delta, C] & \equiv \lambda \text{twin}. \lambda \text{self}. \\
& \{ (\text{Rows}[\Delta, C(X)], 1] \text{ Empty}[C \cdot \text{twin} \cdot m \text{ self})
\end{align*}
\]

Figure 7: Macro for dictionary type.

5 Related Work

This work is based on the Java-to-FLINT translation framework by League, Trifonov, and Shao [7]. We have demonstrated the use of kind polymorphism with type parametrization to represent polymorphism in Java. Compared to previous work such as [2, 1, 3], our target language is arguably more expressive, and also at a lower level than JVML. In our translation, we avoid compromises in type safety by preserving type parameters in the target language. We also avoid unnecessary code expansion by representing generic classes as first-class constructs in the target language. A comparison of relative strengths and weaknesses of previous proposals can be found in the research note [9].

6 Conclusion

We have developed a faithful and efficient encoding of generic types for Java in a simple, implementable typed intermediate language. The two advantages of our translation, as compared to previous proposals, are: (1) Our translation does not erase type information—all type parameters are preserved in the object code; (2) Our translation do not require code expansion to achieve type-preservation. Our main contribution would be that we directly captured type parametricity in a class-based O-O language in a simple and implementable variant of typed lambda calculus. The translation of FGJ can also be naturally extended to a significant subset of Generic Java. An implementation of this translation is planned.

\[\text{A.2 Well-formed types}\]

\[
\begin{align*}
\Delta \vdash \text{X} \in \text{Object ok} \quad & (16) \\
\Delta \vdash \text{X ok} \quad & (17) \\
\Delta \vdash \text{C(T) ok} \quad & (18)
\end{align*}
\]

References


\[\text{A Source Language Semantics}\]

\[\text{A.1 Subtyping}\]

\[
\begin{align*}
\Delta \vdash \text{T}:X \\
\Delta \vdash \text{S}<:\text{T} \\
\Delta \vdash \text{S}:U \\
\Delta \vdash \text{X}:\Delta(\text{X}) \\
\text{CT}(\text{C}) = \text{class C(X)c[N}{\ldots}\} \text{ N} \\
\Delta \vdash \text{C(T)}<:\text{T}[\text{X}]\text{N} \\
\end{align*}
\]

\[\text{A.2 Well-formed types}\]

\[
\begin{align*}
\Delta \vdash \text{X ok} \quad & (16) \\
\Delta \vdash \text{X ok} \quad & (17) \\
\Delta \vdash \text{C(T) ok} \quad & (18)
\end{align*}
\]

Class declaration translation:

\[ cdec[C] = (A_{\text{C}}, \text{Aext}_{\text{C}}) : \cdots : (\Omega \mapsto \Delta) \rightarrow (\Omega \mapsto \text{tail}[l_i]). \) \]

**Constructors code:**

\[ fieldvec(C) = T_1 \ldots T_n \]  

**Method code:**

\[ CT(C) = \text{class } C(\chi \circ \delta) \n N \{ \ldots ; K_{M_1}, \ldots M_n \} \]  

**Method code:**

\[ \Gamma \vdash \delta \vdash \chi \vdash \delta \vdash \text{this}:C(X) \]  

**Method code:**

\[ \text{class } C(\chi \circ \delta) \]  

**Auxiliary definitions**

Bound of type

\[ \text{bound}_{\Delta}(C) = \Delta(X) \]  

**Field lookup**

\[ \text{fields}(\text{Object}) = \cdot \]  

---

A.4 Expression typing

\[ \Delta ; \Gamma ; \chi \vdash X \in \Delta(X) \]  

**Bound of type**

\[ \text{bound}_{\Delta}(N) = \n N \]  

**Field lookup**

\[ \text{fields}(\text{Object}) = \cdot \]  

---

A.3 Computation

\[ \text{fields}(N) = \text{new } N(\emptyset) \rightarrow \emptyset \]  

**Bound of type**

\[ \text{bound}_{\Delta}(N) = \n N \]  

**Field lookup**

\[ \text{fields}(\text{Object}) = \cdot \]  

---

A.7 Auxiliary definitions

**Field lookup**

\[ \text{fields}(\text{Object}) = \cdot \]  

---

Figure 5: Translation of generic class declarations.
Method type lookup

\[
CT(c) = \text{class } C(x : \mathcal{N}) < \mathcal{N} \{ \mathcal{S} \mathcal{F}; K \mathcal{R} \}
\]

\[
\text{mtype}(m, CT) = [T[1], C]\{I \circ \mathcal{O} \mathcal{U}\}\}
\]

(33)

Method body lookup

\[
CT(c) = \text{class } C(x : \mathcal{N}) < \mathcal{N} \{ \mathcal{S} \mathcal{F}; K \mathcal{R} \}
\]

\[
mbody(m, CT) = (x, [T[1], V[1], C]\{\mathcal{O}_\mathcal{U}\})
\]

(35)

Valid method overriding

\[
\text{mbody}(m, \mathcal{V}[1], CT) = \text{mbody}(m, [T[1], C]\{\mathcal{O}_\mathcal{U}\})
\]

(36)

B.2 Kind environment formation

\[
\mathcal{E} \vdash \alpha : \kappa
\]

\[
\mathcal{E} \vdash \circ \text{ type env}
\]

(44)

B.3 Type formation

\[
\mathcal{E} \vdash \Delta : \tau : \kappa
\]

(46)

\[
\mathcal{E} \vdash \Delta : \alpha :: \kappa
\]

(47)

\[
\mathcal{E} \vdash \Delta : \tau_1 :: \kappa \quad \mathcal{E} \vdash \Delta : \tau_2 :: \kappa
\]

(48)

\[
\mathcal{E} \vdash \Delta : \{\tau_1 \ldots \tau_n = \kappa\} :: \{\kappa_1 : \kappa_n\}
\]

(49)

\[
\mathcal{E} \vdash \Delta : \tau : \{\kappa_1 : \kappa_n\}
\]

(50)

B. Target Language Semantics

B.1 Kind formation

\[
\mathcal{E} \vdash \kappa
\]

\[
\mathcal{E} \vdash \Omega
\]

(38)

\[
\mathcal{E} \vdash R^L
\]

(39)

\[
\mathcal{E} \vdash \kappa' \quad \mathcal{E} \vdash \kappa
\]

\[
\mathcal{E} \vdash \kappa' \vdash \kappa
\]

(40)
B.4 Type equivalence

\[
\begin{align*}
\mathcal{E}; \Delta \vdash \tau &\equiv \eta \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta' \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta'' \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-1)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-2)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-3)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-4)}
\end{align*}
\]

(51) – (87)

B.5 Type environment formation

\[
\begin{align*}
\mathcal{E}; \Delta \vdash \tau &\equiv \eta \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta' \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta'' \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-1)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-2)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-3)} \\
\mathcal{E}; \Delta \vdash \tau &\equiv \eta^{(l-4)}
\end{align*}
\]

(51) – (87)