A Recipe for Raw Types

Atsushi Igarashi
University of Tokyo
igarashi@graco.c.u-tokyo.ac.jp

Benjamin C. Pierce
University of Pennsylvania
bcpierce@cis.upenn.edu

Philip Wadler
Avaya Labs
wadler@avaya.com

Abstract

The design of GJ (Bracha, Odersky, Stoutamire and Wadler), an extension of Java with parametric polymorphism, was significantly affected by the issue of compatibility between legacy Java code and new GJ code. In particular, the introduction of raw types made it easier to interface polymorphic code with monomorphic code. In GJ, for example, a polymorphic class List<X>, parameterized by the element type X, provides not only parameterized types such as List<Object> or List<String> but also the raw type List; then, a Java class using List can be compiled without adding element types to where List is used. Raw types, therefore, can reduce (or defer, at least) programmers’ burden of modifying their old Java code to match new polymorphic code.

From the type-theoretic point of view, raw types are close to existential types in the sense that clients using a raw type C expect some implementation of a polymorphic class of the same name C. Unlike ordinary existential types, however, raw types allow several unsafe operations such as coercion from the raw type List, whose element type is abstract, to List<T> for any concrete type T. In this paper, basing on Featherweight GJ, proposed by the authors as a tiny core language of GJ, we formalize a type system and direct reduction semantics of raw types. The bottom type, which is subtype of any type, plays a key role in our type-preserving reduction semantics. In the course of the work, we have found a flaw in the typing rules from the GJ specification; type soundness is proved with respect to a repaired version of the type system.

1 Introduction

In the past few years, a number of extensions of Java with parametric polymorphism have been proposed by various groups [1, 13, 14, 3, 7, 17] with various design and implementation schemes. Among them, the design of GJ [3], proposed by Bracha, Odersky, Stoutamire, and Wadler, is significantly affected by issues concerning compatibility with the current Java language environment. In particular, the language is backward compatible with Java in the sense that every Java program is also a GJ program, and forward compatible in the sense that GJ programs are compiled to Java Virtual Machine Language so that they can run on the Java environment.

Among the features of GJ, raw types are designed to help program evolution, the process by which monomorphic programs are upgraded to polymorphic ones. For example, suppose we have the following two Java classes, Pair and Client:

```java
class Pair {
    Object fst; Object snd;
    Pair(Object fst, Object snd){ this.fst=fst; this.snd=snd; }
    void setfst(Object newfst){ fst=newfst; }
}
class Client {
    Object getfst(Pair p){ return p.fst; }
    void setfst(Pair p, Object x){ p.setfst(x); }
}
```

Then, we happen to decide to replace the old language (Java) with a new language (GJ) and rewrite the monomorphic Pair class to a polymorphic version below:
The class Pair takes two type parameters X and Y; by instantiating them with arbitrary (reference) types, we can use the above definition as a class for pairs of specific element types: pairs of strings (Pair<String,String>), pairs of integers (Pair<Integer,Integer>), and so on. Under an ordinary type system, however, the two classes Client and Pair<X,Y> together would not be well typed any more because the class Client does not use the class Pair<X,Y> as is expected—for example, clearly, type arguments are missing everywhere Pair is mentioned in Client. Thus, one would have to rewrite all the classes that use Pair, making it hard to evolve old code gradually.

In GJ, every parameterized class C<X₁,...,Xₙ> provides the raw type C, supertype of any parameterized type from the class such as C<Object,... > or C<String,... >. The field and method types of a raw type are obtained by erasure of the original definition: all type arguments are dropped and type variables are promoted to Object. For example, the fields fst and snd of the raw type Pair are given type Object and the method setfst of Pair is given type Object→Pair. (Notice that these types are the same as ones from the old monomorphic Pair class.) Hence, the class Client will remain well typed even when used together with the new polymorphic Pair class.

Furthermore, in order to make it even easier for legacy Java code to coexist with polymorphic code, GJ permits several unsafe operations such as coercion from a raw type to a parameterized type. For example, not only can an expression of Pair<String,String> be passed to where Pair is expected, but also an expression of Pair can be passed to where Pair<Integer,Integer> is expected. The latter coercion is clearly unsafe because an expression of Pair is not necessarily a pair of integers. The GJ compiler accepts such unsafe operations, signaling unchecked warnings. In this sense, GJ does not guarantee static type safety of programs under evolution; dynamic safety is, however, still guaranteed because unchecked programs are compiled to well-typed bytecode and so only reasons of failure will be due to downcasts that the GJ compiler inserted. Such relaxation seems preferable (or even required) for smooth evolution: in fact, a lot of practical examples, including the one above, are classified unchecked. Since they are not statically safe, unchecked programs are not supposed for permanent use and are expected to evolve eventually into typesafe (warning-free) programs by augmenting raw types with appropriate type arguments. The introduction of raw types, nevertheless, makes it possible to reduce or, at least, significantly defer the burden of modifying old Java code so that it matches with the evolved polymorphic code.

The specification document of GJ [2] explains what raw types are and when unchecked warnings should be signaled along with its compilation scheme. This specification and the companion paper [3] give clear basic intuitions about the behavior of raw types, but it falls short of a fully formal account. It is written in English prose and the description is largely given in terms of compilation into Java. It would be desirable to supplement the prose with a formal description and to replace the description in terms of compilation with a direct specification of the semantics. Formal direct description would also help to apply the idea of raw types to other generic extensions of Java or even to other programming languages.

The goal of this work is to provide a formal description of raw types. As a first step, this paper focuses on formal accounts on a direct semantics of a language with raw types and a type system of raw types. (We also aim to discuss their role in program evolution, but, for the moment, leave its formal treatment for future work). Our main contributions here are summarized as follows:

- Formalization of a type system for a core of GJ with raw types. The type system accepts unsafe operations on raw types but signals unchecked warnings against them; thus, correct classification of safe and unsafe operations is critical for the type system.

  From the type-theoretic point of view, raw types are close to existential types [12] in the sense that clients using a raw type C expect some implementation of a polymorphic class of the same name C. This informal connection turns out to be helpful to obtain intuitions why certain operations should be unsafe.

- Formal operational semantics. We give for the first time a direct reduction semantics of raw types.
The semantics is direct in the sense that it does not depend on translation into lower-level languages such as Java; type argument passing is expressed in the reduction relation just as in System F [8, 16].

One technical challenge was in developing rules for method invocation on raw types. The type system assumes that raw types does not involve polymorphic methods; so every method invocation \( e.m(t) \) where \( e \) is of raw type, say \( C \), is well typed without type arguments. At run-time, however, \( e \) can reduce to an object new \( C<T>(\bar{\mathcal{I}}) \) of parameterized type, which might contain polymorphic methods; then, some appropriate types for polymorphic methods have to be ‘made up’ and passed as type arguments. For this purpose, we introduce the bottom type, which is subtype of every type, into the language; the bottom type is passed for such missing type arguments.

- Proof of type soundness. We prove that programs without unchecked warnings (called checked programs) are indeed typesafe. In the course of a proof, we have discovered one serious flaw in the typing rules given in the original specification; a repaired version of the type system is proved to be sound.

The basis of our work is a core calculus called Featherweight GJ, or FGJ [10]. This calculus was originally developed as an extension of Featherweight Java (or FJ) [10], which was designed to investigate consequences of complex class-based extensions such as polymorphic classes and inner classes [9]. FJ and FGJ were designed to omit as many features of Java and GJ as possible (even assignment), while maintaining the essential flavor of the language and its type system. As a result, the complex FGJ definition fits comfortably on a few pages, and its basic properties can be proved with no more difficulty than, say, those of the polymorphic lambda-calculus with subtyping (such as System \( F_{\leq} \) [5]). This simplicity makes it feasible to give formal accounts on raw types.

The rest of the paper is organized as follows. First, Section 2 reviews our basic calculus Featherweight GJ; the following section extends Featherweight GJ with raw types and develops Raw FGJ; we also show a theorem of type soundness. After discussing related work in Section 4, Section 5 gives concluding remarks with directions to future work. For brevity, proofs of theorems are omitted; they will appear in a forthcoming technical report [11].

2 Featherweight GJ

We begin by reviewing the basic definitions of Featherweight GJ (FGJ, for short) [10]. FGJ is a tiny fragment of GJ, including only top-level class definitions, object instantiation, field access, method invocation, and typecasts.

2.1 Syntax

The abstract syntax of FGJ types, class declarations, constructor declarations, method declarations, and expressions is given at the top left of Figure 1. The metavariables \( A, B, C, D, \) and \( E \) range over class names; \( S, T, U, \) and \( V \) range over types; \( X, Y, \) and \( Z \) range type variables; \( \mathcal{N}, \mathcal{P}, \) and \( Q \) range over non-variable types; \( L \) ranges over class declarations; \( K \) ranges over constructor declarations; and \( H \) ranges over method declarations; \( f \) and \( g \) range over field names; \( m \) ranges over method names; \( x \) ranges over variables; and \( e \) and \( d \) range over expressions. We write \( \bar{F} \) as shorthand for a possibly empty sequence \( f_1, \ldots, f_n \) (and similarly for \( \bar{C}, \bar{X}, \bar{P},\) etc.) and write \( \bar{Y} \) as shorthand for \( Y_1 \ldots Y_n \) (with no commas). We write the empty sequence as \( \cdot \) and denote concatenation of sequences using a comma. The length of a sequence \( \bar{x} \) (or \( \bar{X} \)) is written \#(\( \bar{x} \)) (or \#(\( \bar{X} \)), respectively). We abbreviate operations on pairs of sequences in the obvious way, writing \( \bar{C} \bar{F} \) as shorthand for “\( C_1 \bar{f}_1, \ldots, C_n \bar{f}_n \)” and “\( \bar{C} \bar{F} \)” as shorthand for “\( C_1 \bar{f}_1, \ldots, C_n \bar{f}_n \)” and “\( \text{this} \bar{F} \)” as shorthand for “\( \text{this}.f_1=\bar{f}_1; \ldots, \text{this}.f_n=\bar{f}_n; \)” and “\( \bar{X}\bar{P} \)” as shorthand for “\( X_1 N_1, \ldots, X_n N_n \).” Sequences of field declarations, parameter types, variable types, and method declarations are assumed to contain no duplicate names. As in GJ, the empty brackets \( <> \) are omitted.

A class declaration consists of its name (class \( C \)), type parameters (\( \bar{X} \)) with their bounds (\( \bar{\mathcal{N}} \)), fields (\( \bar{F} \bar{P} \)), one constructor (\( \bar{K} \)), and methods (\( \bar{R} \)); moreover, every class must explicitly declare its supertype \( \mathcal{N} \) with \( < \) (abbreviation of extends) even if it is Object. The bound of a type variable may not be a type variable, but may be a type expression involving type variables, and may be recursive (or even, if
Syntax:

\[ T(S, U, V) \text{ (types)} \]
\[
\begin{align*}
  &::= X & \text{type variables} \\
  &\quad | N & \text{non-variable types}
\end{align*}
\]

\[ N(P, Q) \]
\[
\begin{align*}
  &::= C<T>
\end{align*}
\]

\[ L \text{ (class declarations)} \]
\[
\begin{align*}
  &::= \text{class } C<\vec{X}<\vec{N}>  \text{ (T; } K \vec{M}) \\
  K \text{ (constructors)} &::= C(T)\{ \text{ super(T); this.}T=; \} \\
  M \text{ (methods)} &::= <\vec{X}<\vec{N}>T \text{ m (T } \vec{X}\} \{ \text{ return e; } \}
\]

\[ e \text{ (expressions)} \]
\[
\begin{align*}
  &::= x & \text{variables, this} \\
  &\quad | e.f & \text{field access} \\
  &\quad | e.<T>(\pi) & \text{method invocation} \\
  &\quad | \text{ new } N(\pi) & \text{object instantiation} \\
  &\quad | (N)e & \text{typecast}
\end{align*}
\]

Bound of type:

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

Subclassing:

\[ C \subseteq C \]
\[ C \subseteq D \quad D \subseteq E \]
\[ C \subseteq E \]

\[ CT(C) = \text{class } C<\vec{X}<\vec{N}> \text{ (T;} \vec{D}; \vec{T}) \{ \ldots \} \]
\[ C \subseteq D \]

Field lookup:

\[ \text{fields}(\text{Object}) = \ast \]
\[ CT(C) = \text{class } C<\vec{X}<\vec{N}> \text{ (T; K } \vec{M}) \]
\[ \text{fields}(T/X)N = U Y \]

Method type lookup:

\[ CT(C) = \text{class } C<\vec{X}<\vec{N}> \text{ (T; K } \vec{M}) \]
\[ \text{mtype}(m, C<T>) = [T/X](<Y<F>U\rightarrow U) \]

Method body lookup:

\[ CT(C) = \text{class } C<\vec{X}<\vec{N}> \text{ (T; K } \vec{M}) \]
\[ \text{mbody}(m<T>, C<T>) = (x, T/X, V/Y)e_0 \]

Valid downcast:

\[ \text{dcast}(S, T) \quad \text{dcast}(T, U) \]
\[ \text{dcast}(S, U) \]

\[ CT(C) = \text{class } C<\vec{X}<\vec{N}> \text{ (T;} \vec{D}; \vec{T}) \{ \ldots \} \]
\[ FV(N) \]
\[ \text{dcast}(C<T>, T/X)N \]

\[ (FV(N) \text{ denotes the type variable in } N.) \]

Figure 1: FGJ: Syntax and Auxiliary Definitions
there are several bounds, mutually recursive); in this sense, FGJ supports an extended form of F-bounded polymorphism [4]. Each argument of a constructor corresponds to an initial (and also final) value of each field of the class. As in Java and GJ, fields inherited from superclasses are initialized by super(T); and newly declared fields by this.f=T; although, as we will see, those statements do not have significance during execution of programs—this syntax is adopted just to keep FGJ as close as GJ (and Java). A body of a method just returns an expression, which is either a variable, field access, method invocation, object instantiation, or typecast. Unlike GJ, which infers type arguments for polymorphic methods, FGJ requires explicit type arguments T for method invocation e.m<T>(w). We treat this in method bodies as a variable, and so require no special syntax. As we will see later, the typing rules prohibit this from appearing as a method parameter.

A class table CT is a mapping from class names C to class declarations L; a program is a pair (CT,e) of a class table and an expression. Object is treated specially in every FGJ program: the definition of Object class never appears in the class table and the auxiliary functions that look up field and method declarations in the class table are equipped with special cases for Object that return the empty sequence of fields and the empty set of methods. (As we will see later, method lookup for Object is just undefined.) To lighten the notation in what follows, we always assume a fixed class table CT.

The given class table is assumed to satisfy some sanity conditions: (1) CT(C)=class C... for every C ∈ dom(CT); (2) Object ∈ dom(CT); (3) for every class name C (except Object) appearing anywhere in CT, we have C ∈ dom(CT); and (4) there are no cycles in the reflexive and transitive closure of the relation between class names induced by < clauses in CT. (The formal definition of that relation ⊆ appears in Figure 1.) By the condition (1), we can identify a class table with a sequence of class declarations in the customary manner. The body of the method invocation m at nonvariable type N is a type of the form <X><N>U→U0, where X are type parameters, U are argument types, and U0 is the result type. In this form, the variables X are bound in N, U, and U0 and we regard α-convertible ones as equivalent; application of type substitution [T/X] is defined in the customary manner. The body of the method invocation m at nonvariable type N with type parameters Y, written mbody(m<Y>,N), is a pair, written (X,e), of a sequence of parameters X and an expression e. (Note that the functions mtype(m,N) and mbody(m<Y>,N) are both partial functions: since Object is assumed to have no methods in FGJ, both mtype(m,Object) and mbody(m<Y>,Object) are undefined.) We write m ∉ X to mean the method definition of the name m is not included in X.

2.2 Type System

The type system of FGJ consists of three forms of judgments: one for subtyping Δ ⊢ S <: T, one for type well-formedness Δ ⊢ e : T, and one for typing Δ;Γ ⊢ e ∈ T, where Γ is an environment, a finite mapping from variables to types, written ξ:ξ. Figure 2 shows the rules to derive these judgments. We abbreviate a sequence of judgments, writing Δ ⊢ S <: T as shorthand for Δ ⊢ S1 <: T1, ..., Δ ⊢ Sn <: Tn and Δ ⊢ T ok as shorthand for Δ ⊢ T1 ok, ..., Δ ⊢ Tn ok and Δ;Γ ⊢ e ∈ T as shorthand for Δ;Γ ⊢ e1 ∈ T1, ..., Δ;Γ ⊢ en ∈ Tn.

Subtyping is the reflexive and transitive closure of the relation induced by <: clauses. Type parameters are invariant with regard to subtyping (for the usual reasons: type parameters can be used both positively and negatively), so Δ ⊢ T <: U does not imply Δ ⊢ S <: T <: U.

If the declaration of a class C begins with class C<Y>, then a type like C<Y> is well formed only if substituting T for Y respects the bounds X, that is if Δ ⊢ T <: [T/X]X. We write Δ ⊢ T ok if type T is well-formed in context Δ. Note that we perform a simultaneous substitution, so any variable in X may appear in Y, permitting recursion and mutual recursion between variables and bounds. A type environment Δ is well formed if Δ ⊢ Δ(X) ok for all X in dom(Δ).

The typing rules are syntax directed, with one rule for each form of expression, save that there are three rules for casts. The typing rules for constructor/method invocation check that each actual parameter is given subtype of the corresponding formal. There are three rules for type casts: in an upcast the subject
Subtyping:

\[ \Delta \vdash X : \Delta(X) \]

Well-formed types:

\[ \Delta \vdash \text{Object} \quad \Delta \vdash X \in \text{dom}(\Delta) \]

Expression typing:

\[ \Delta; \Gamma \vdash x \in \Gamma(x) \]

Method typing:

\[ mtype(m, bound) = \text{class } C<\text{X} \triangleright N \triangleright N \{ \ldots \} \]

Class typing:

\[ \text{fields}(N) = \text{class } C<\text{X} \triangleright N \triangleright N \{ \text{super(}\text{G}; \text{this.}\text{T}; \text{K}; \text{N}); \text{OK} \} \]

Computation:

\[ \text{fields}(N) = \text{new } N(\text{G}); \text{f, i } \rightarrow \text{e}, \]

\[ \text{mbody}(m, X, N) = (X, e_0) \]

\[ \text{new } N(\text{G}).m, X, (d) \rightarrow [d/X, \text{new } N(\text{G})/\text{this}]e_0 \]

\[ \varnothing \vdash N : P \]

\[ (P)(\text{new } N(\text{G})) \rightarrow \text{new } N(\text{G}) \]

Figure 2: FGJ: Subtyping, Type Well-Formedness, Typing and Reduction Rules
is subtype of the target, in a downcast the target is (strict) subtype of the subject, and in a stupid cast the target is unrelated to the subject. The side condition $\text{decast}(S, T)$ in the rule for downcasts ensures that the result of the cast will be the same at run time, no matter whether we use the high-level (type-passing) reduction rules defined later in this section or the erasure semantics (implemented by the current compiler and defined in our previous paper [10].) Since type arguments are removed in the erasure semantics, a downcast $(C<\forall>T)e$ is allowed only if all type arguments $\forall$ 'contribute' to decide $\Delta \vdash C<\forall>T : bound_\Delta(T)$ where $T$ is the type of $e$—the condition is expressed by $X = FV(N)$ in the definition of $\text{decast}(S, T)$. For example, $(\text{Pair}<A,B>e$ where $e$ is given type Object should not be accepted as type arguments $A$ and $B$ are lost in the erasure semantics. Stupid casts are ones that certainly fail when evaluated; they are required to formulate type soundness through a subject reduction theorem for a small-step semantics: a downcast may reduce eventually to a stupid cast. Unlike GJ compiler, which rejects an expression containing stupid casts as ill-typed, FGJ indicate the special nature of stupid casts by including the hypothesis stupid warning in the typing rule for stupid casts. See [10] for detailed discussions on the rules for typecasts.

The typing judgment for method declarations has the form $M \text{ OK in } C<\forall>T$, read “method declaration $M$ is ok if it occurs in class $C<\forall>T$.” It uses the expression typing judgment on the body of the method, where the free variables are the parameters of the method with their declared types, plus the special variable this with type $C<\forall>T$. (Thus, a method with a parameter of name this is disallowed as the type environment is ill formed.) In case of overriding, if a method with the same name is declared in a superclass, then the two methods must have the same argument types and the same type variables with the same bounds (modulo $\alpha$-conversion); the result type of the overriding method may be a subtype of the result type of the overridden—the predicate override checks the condition. Note that, unlike Java, FGJ (and GJ) allow covariant overriding [10, 3]. The typing judgment for class declarations has the form $L \text{ OK in } C<\forall>T$, read “class declaration $L$ is ok.” It checks that the constructor applies super to the fields of the superclass and initializes the fields declared in this class, and that each method declaration in the class is ok.

2.3 Reduction Semantics

The reduction relation is of the form $e \rightarrow e'$, read “expression $e$ reduces to expression $e'$ in one step.” We write $\rightarrow^*$ for the reflexive and transitive closure of $\rightarrow$. The reduction rules are given at the bottom right of Figure 2. There are three reduction rules, one for field access, one for method invocation, and one for typecast. Field access new $N(\forall).f_i$ looks up and obtains field names $\forall$ of $N$ with fields($N$); then it reduces to the constructor argument $e_i$ of the corresponding position. Method invocation new $N(\forall).m(\overline{d})$ first looks up $\text{mbody}(m, N)$ and obtains a pair of a sequence of formal arguments $\forall$ and the method body; then, it reduces to the method body in which $\forall$ are replaced with the actual arguments $\overline{d}$ and this with the receiver new $N(\forall)$. We write $[d_0/\forall, e/y]_0$ to stand for replacing $x_1$ by $d_1$, $\ldots$, $x_n$ by $d_n$, and $y$ by $e$ in the expression $e_0$. Typecast ($P$)new $N(\forall)$ removes $P$ if the type $N$ of the object is subtype of the target $P$ of the cast. If not, then no rule applies and the computation is stuck, denoting a run-time error. As we already mentioned above, the type system is designed to make the direct semantics and the erasure semantics that the current compiler uses coincide, though the direct semantics shown here can potentially express type-dependent operations such as downcasts that need run-time type argument information.

The reduction rules may be applied at any point in an expression, so we also need the obvious congruence rules (if $e \rightarrow e'$ then $e.f \rightarrow e'.f$, and the like), which are omitted for brevity.

For example, under the class table including the following class definitions

```
class A extends Object { A(){ super(); } }
class B extends A { B(){ super(); } }
class Pair<X extends Object, Y extends Object> extends Object {
    X fst; Y snd;
    Pair(X fst, Y snd) { super(); this.fst=fst; this.snd=snd; }
    Pair<X,Y> setfst(X newfst) { return new Pair<X,Y>(newfst, this.snd); }
}
```

the expression new $\text{Pair}<A,B>(\text{new } A(), \text{ new } B()).\text{setfst}(\text{new } B()).\text{snd}$ evaluates to new $B()$ via the
following reduction steps (where the next subexpression to be reduced is underlined at each step):

\[
\text{new Pair}\langle A, B \rangle (\text{new A()}, \text{new B()}).\text{setfst}(\text{new B()}).\text{snd} \\
\rightarrow \text{A/X, B/Y, new B()}/\text{newfst, } \text{new Pair}\langle A, B \rangle (\text{new A()}, \text{new B()}/\text{this}) \rightarrow \text{new Pair}\langle X, Y \rangle (\text{newfst, this snd}) \\
= \text{new Pair}\langle A, B \rangle (\text{new B(), new Pair}\langle A, B \rangle (\text{new A()}, \text{new B()}).\text{snd}).\text{snd} \\
\rightarrow \text{new Pair}\langle A, B \rangle (\text{new B(), new B()}).\text{snd} \\
\rightarrow \text{new B()}
\]

2.4 Properties

FGJ is type sound, shown by the standard technique [18] using subject reduction and progress properties [10].

2.4.1 Theorem [FGJ subject reduction]: If \( \Delta; \Gamma \vdash e \in T \) and \( e \rightarrow e' \), then \( \Delta; \Gamma \vdash e' \in T' \), for some \( T' \) such that \( \Delta \vdash T' <: T \).

2.4.2 Theorem [FGJ progress]: Suppose \( e \) is a well-typed expression.

1. If \( e \) includes \( \text{new } N_0(\overline{\sigma}).f \) as a subexpression, then \( \text{fields}(N_0) = \overline{T} \overline{T} \) and \( f \in \overline{T} \).

2. If \( e \) includes \( \text{new } N_0(\overline{\sigma}).m<_V(\overline{d}) \) as a subexpression, then \( \text{mbody}(m<_V, N_0) = (x, e_0) \) and \( \#(x) = \#(d) \).

2.4.3 Theorem [FGJ type soundness]: If \( \emptyset; \Gamma \vdash e \in T \) and \( e \rightarrow^* e' \) with \( e' \) being a normal form, then \( e' \) is either (1) a value \( v \) (given by the syntax \( v ::= \text{new } N(\overline{\sigma}) \)) with \( \emptyset; \Gamma \vdash v \in S \) and \( \emptyset \vdash S <: T \) or (2) an expression containing \( (P)\text{new } N(\overline{\sigma}) \) where \( \emptyset \vdash N \not\in P \).

3 Raw FGJ

Now, we develop the language Raw FGJ, by extending FGJ with raw types. We begin with a simple example to see how raw types behave and what are technical subtleties in Raw FGJ.

3.1 Raw Types and Unsafe Operations

Consider the class \( \text{Pair}\langle X, Y \rangle \) from the last section and the following class \( \text{Client} \), rewritten in Raw FGJ, from the introduction.

```java
class Pair<X extends Object, Y extends Object> extends Object {
    X fst; Y snd;
    Pair(X fst, Y snd) { super(); this.fst=fst; this.snd=snd; }
    Pair<X,Y> setfst(X newfst) { return new Pair<X,Y>(newfst, this.snd); }
}

class Client extends Object {
    Client() { super(); }
    Object getfst(Pair p){ return p.fst; }
    Pair setfst(Pair p, Object x){ return p.setfst(x); }
}
```

Raw types provide erased field and method types: the fields \( \text{fst} \) and \( \text{snd} \) are given type \( \text{Object} \), which is the bound of \( X \) and \( Y \), and the constructor takes a pair of \( \text{Objects} \). The method \( \text{setfst} \) (from \( \text{Pair} \)) is given type \( \text{Object} \rightarrow \text{Pair} \), in which type parameters are removed from the result type \( \text{Pair}\langle X, Y \rangle \). Concerning subtyping, any parameterized type \( \text{Pair}\langle S, T \rangle \) is subtype of the raw type \( \text{Pair} \).

The following operations are permitted but considered unsafe (in both GJ and Raw FGJ) and GJ compilers are supposed to signal unchecked warnings against them:

- Coercion from a raw type \( C \) to a parameterized type \( C<T> \). For example, passing an expression of raw type \( \text{Pair} \) to a method expecting an argument of \( \text{Pair}<\text{String}, \text{String}> \) is permitted but signals an unchecked warning.
• Method/constructor call on a raw type, if erasure changes the argument types. (The result type of
the method may be changed.) For example, the invocation of \texttt{setfst} on \texttt{Pair} is unchecked because
the argument type is changed by erasure from \(X\) to \texttt{Object}. Similarly, \texttt{new Pair(e1,e2)} (without type
arguments) is unchecked.

(In GJ, an assignment to a field is unchecked if erasure changes the field type; this rule is not directly modeled
here because FGJ omits assignments. A ‘setter’ method like \texttt{setfst}, however, results in a similar restriction:
a field type appears as an argument type, being unchecked by the second rule above.) Thus, in fact, the
two classes \texttt{Pair<X,Y>} and \texttt{Client} together are unchecked because of the invocation of \texttt{setfst} in \texttt{Client}.
When a raw type is specified as the superclass, access to members defined in superclasses is also considered
access to a raw type. For example, in the following declaration, invocation of \texttt{setfst} is unchecked.

```java
class Foo extends Pair {
    Foo(Object fst, Object snd) { super(fst, snd); }
    Object id (Object x) { return this.setfst(x).fst; }
}
```

In the specification document, however, there is only a few examples that show how unchecked programs
might fail and no clear intuitive explanation why they are potentially dangerous.

### 3.2 Correspondence between Raw Types and Existential Types

From the type-theoretic point of view, raw types resemble (bounded) existential types \([12, 6]\) (with certain
unsafe operations permitted) in the sense that clients using a raw type \(C\) expect \textit{some} implementation of a
polymorphic class of the same name \(C\). By this analogy, for example, the raw type \texttt{Pair} would be considered
the existential type \(\exists X<:Object,Y<:Object.Pair<X,Y>\).

Then, the basic typing rules of raw types can be explained as follows: every access to a raw type would
involve implicit existential unpacking and, if the type of the whole expression involves an existential
type variable, then it is promoted to a supertype, without the variable. For example, \(p.fst\) would be given type \(X\),
then promoted to \texttt{Object}; similarly \(p.setfst(x)\) would be given type \texttt{Pair<X,Y>}, promoted to the raw type
\texttt{Pair}. Actually, coercion from a parameterized type to a raw type would be considered existential packing.
For example, passing an expression \(e\) of type \texttt{Pair<A,B>} to where \texttt{Pair} is expected roughly corresponds to

```java
pack [X=A,Y=B;e] as \exists X<:Object,Y<:Object.Pair<X,Y>.
```

Thus, the raw method invocation \(p.setfst(x)\) would be considered a combination of unpacking and method
invocation, followed by packing of the result from the invocation.

Unsafe operations listed above are also explained in terms of this analogy. First, coercion from raw types
to parameterized types is clearly unsafe: one cannot safely instantiate existential type variables by concrete
types. Second, since the typing rules of Raw FGJ (and GJ) approximate the method argument types, by
erasure, to their supertypes, raw method invocation is safe only if the erased argument types are equal to the
original. For example, consider the existential type \(\exists X<:Object,Y<:Object.Pair<X,Y>\) again; the method
\texttt{setfst} would be given type \(X\rightarrow Pair<X,Y>\) but, since we do not know the identity of \(X\), we can neither
approximate \(X\) by \texttt{Object} nor use the method in a type-safe manner. In fact, with the following subclass
\texttt{IdPair} of \texttt{Pair}

```java
class Id extends Object {
    Id() { super(); }
    Id id() { return this; }
}
class IdPair extends Pair<Id,Id> {
    IdPair(Id fst, Id snd) { super(fst, snd); }
    IdPair setfst(Id newfst) { return new IdPair(newfst.id(), this.snd); }
}
```

the expression

```java
new Client().setfst(new IdPair(new Id(), new Id()), new Object())
```
fails at the method invocation \texttt{newfst.id()}. A similar argument applies to raw constructor invocation: for example, with the following class:

\begin{verbatim}
class D<X extends Object> extends Object {
   Pair<Id,Id> g;
   D(Pair<Id,Id> g) { super(); this.g = g; }
   Id loophole() { return this.g.fst.id; }
}
\end{verbatim}

the expression

\begin{verbatim}
   new D(new Pair<Object,Id>(new Object(), new Id())).loophole()
\end{verbatim}

fails at the method invocation \texttt{this.g.fst.id}, too. Notice that the argument type of the raw constructor \texttt{D} is changed from \texttt{Pair<Id,Id>} to \texttt{Pair} by erasure, thus accepting an object of \texttt{Pair<Object,Id>} as the argument. Actually, the GJ specification overlooks the necessity of this check at raw constructor calls and the current compiler (version 0.6m) does not signal an unchecked warning against any raw constructor invocation.\footnote{The dummy type parameter \texttt{X} is required when readers try this example with the GJ compiler: the GJ compiler would identify the type expression \texttt{D} with the parameterized type \texttt{D<>} when the class \texttt{D} is declared without type parameters. Declaring \texttt{D} with type parameters makes sure that \texttt{D} is raw.}

### 3.3 Raw Method/Constructor Invocation and the Bottom Type

As in FGJ, we will formalize an operational semantics of Raw FGJ as type-passing reduction: actual type arguments for a polymorphic method are substituted for the type variables. In Raw FGJ, however, there is one subtlety arising from subtyping between parameterized and raw types. For example, consider the following class \texttt{List} implementing a polymorphic method to obtain a null list for an arbitrary element type:

\begin{verbatim}
class List<X extends Object> extends Object {
   ...
   <Y extends Object> List<Y> makenull() { return new Null<Y>(); }
}
\end{verbatim}

\begin{verbatim}
class Null<X extends Object> extends List<X> { ... }
\end{verbatim}

Since the type system erases even type parameters of methods, method invocation \texttt{e.makenull()}, where \texttt{e} is of raw type \texttt{List}, is well typed without type arguments. (It is also checked invocation; note that erasure does not change the argument types, which are empty.) At run-time, however, \texttt{e} may eventually reduce to an object \texttt{new List<T>(d)} of parameterized type; then, it is required to ‘make up’ appropriate type arguments for the type parameter \texttt{Y} of \texttt{makenull}.

We introduce the bottom type \(*\), which is subtype of any type, for missing type arguments; method invocation lacking actual type arguments reduces to the method body where all the type variables of the method are replaced with the bottom type. For example, \texttt{new List<T>(null).makenull()} reduces to \texttt{new Null<*>().} Passing the bottom type works for the following reasons:

- The bottom type is always lower than the bound of a type variable. Then, the method body remains well typed under the assumption that each formal parameter is given type in which the bottom type is substituted for type variables.

- The types of actual arguments remain compatible with the current method argument types, obtained by substituting the bottom type for type parameters. Since the receiver object was initially given raw type, the method argument types were the erasures of the declared types. However, when the method invocation is checked, the declared formal parameters’ types does not involve type parameters; thus the erasures and the current method argument types should coincide. Then, actual arguments can be safely replaced with formal parameters. Notice that, if type variables appear in types of formal parameters, the types of actual arguments may or may not be compatible after substituting the bottom type.
The context of raw method invocation does not put an assumption about type arguments passed to the method: the result type is always erased to raw—for example, \( e.makenull() \) is given type \( \text{List} \). Hence, returning an expression of parameterized type (for example, \( \text{Null<*>} \), which is subtype of \( \text{List} \)) poses no problem.

Similarly to method invocation, instantiation of raw type, say \( \text{new Pair}(e_1, e_2) \), could be considered constructor invocation missing type arguments for \( X \) and \( Y \). According to this view, unlike the GJ specification, we give a raw constructor invocation \( \text{new Pair}(e_1, e_2) \) type \( \text{Pair<*,*>} \), rather than \( \text{Pair} \), as if a pair of the bottom types were actually passed to the constructor. For a similar reason, when a raw type, say \( C \), is extended by the subclass \( D<X> \), any parameterized type \( D<T> \) from the subclass is subtype of \( C<*,\ldots,*> \) rather than \( C \). This refined type information on raw constructor invocation is required to prove type soundness through subject reduction. Roughly speaking, if we gave raw constructor invocation a raw type, a reduction step involving method invocation would not preserve (checked) well-typedness. For example, a method in a class \( C<X extends N> \) is typed under the assumption that \( \text{this} \) is given type \( C<X> \) with \( X<:N \). Then, for invocation of the method to be type-preserving, \( \text{this} \) must be replaced with an expression of parameterized type; in other words, if the receiver object were given type \( C \), its type would be unsafe subtype of \( C<X> \), the type of \( \text{this} \). Note that, when a raw constructor call is checked, substituting the bottom type will not affect the constructor argument types.

### 3.4 Formal Definition of Raw FGJ

Now, we proceed to the formal definition of Raw FGJ. The full definition of Raw FGJ is given in Figures 3, 4, 5, 6, and 7, where additional rules and changes from FGJ rules are shaded. We mainly focus on the key rules relevant to raw types in what follows.

#### Syntax and Auxiliary Definitions

We use the same notational conventions as in FGJ. A non-variable type \( N \) is now either \textit{cooked}, which has type parameters, or \textit{raw}; the bottom type \( * \) is included in the set of types. We often use \( \bar{T} \) for a sequence of the bottom types \( *,\ldots,* \) (without subscripts). Concerning expressions, method invocation \( e.m(T) \) on raw types is introduced. We enforce to the class table another sanity condition, as well as the ones for FGJ, that the bottom type never appears in \( CT \); the bottom type \( * \) can appear only in the expression to be executed (the second element of a program \( CT, e \)). Unlike GJ, Raw FGJ distinguishes a cooked type \( C<> \) with zero type arguments and the corresponding raw type \( C \). We, however, still abbreviate a class definition beginning with \textit{class} \( C<>N \) \ldots to \textit{class} \( C<N \) \ldots and a method definition \( <>T m(T \bar{T}) \ldots \) to \( T m(T \bar{T}) \ldots \) for conciseness. (GJ is in favor of the cooked type \( C<> \) rather than the raw type \( C \) when you omit \( <> \).

---

**Figure 3: Raw FGJ: Syntax**

- **T(S,U,V)** (types)
  - ::= \( X \) (type variables)
  - | \( N \) (non-variable types)
  - | \( * \) (bottom type)

- **N(P,Q)**
  - ::= \( \text{C}<T> \) (cooked types)
  - | \( \text{C} \) (raw types)

- **L** (class declarations)
  - ::= \text{class} \( C<X<\bar{T}>\ni\ni\ni N \{T \bar{T}; X \bar{R}\} \)

- **K** (constructors)
  - ::= \( C(T \bar{T}) \{ \text{super}(T); \text{this}.T=T; \} \)

- **M** (methods)
  - ::= \( \langle X<\bar{N}> \rangle T m(T \bar{T}) \{ \text{return} e; \} \)

- **e** (expressions)
  - ::= \( x \) (variables, this)
  - | \( e.f \) (field access)
  - | \( e.m(T)(\bar{f}) \) (cooked method invocation)
  - | \( e.m(T) \) (raw method invocation)
  - | \( \text{new N}(\bar{T}) \) (object instantiation)
  - | \( (N)e \) (typecasts)
We require the definition of erasure \(|T|_\Delta\) of type \(T\) under \(\Delta\), defined as the least raw supertype. Note that both \(\text{bound}_\Delta(*)\) and \(|*|_\Delta\) are undefined.

\[
\begin{align*}
\text{head}(C<\mathcal{T}>) &= \text{head}(C) = C \\
\text{|T|_\Delta} &= \text{head}(\text{bound}_\Delta(T))
\end{align*}
\]

The definition of fields(\(N\)) requires the additional rule below for raw types; fields(\(C\)) returns the erased types of the fields.

\[
\begin{align*}
\text{fields}(N) &= U \mathcal{F} \\
\text{fields}(C) &= \{U, Y; C<\mathcal{T}>\}
\end{align*}
\]

3.4.1 Example: Under a class table including class Pair and IdPair,

\[
\begin{align*}
\text{fields(Pair)} &= \{X<\text{Object, Object}, Y<\text{Object, Object}, \text{fst}, \text{snd}\} \\
&= \{\text{Object}, \text{Object}\}
\end{align*}
\]

Notice that types \(\mathcal{F}\) of the fields inherited from subclasses are erased under the type environment \(\mathcal{X}<\mathcal{N}\) from the current class. Hence, the field types of a supertype may or may not be a prefix of ones from a subtype. For example, fields(IdPair) = Id fst, Id snd \(\neq\) Object fst, Object snd = fields(Pair).

In addition to \(\text{mtype}\) and \(\text{mbody}\), we define (partial) functions \(\text{mtyperaw}\) and \(\text{mbodyraw}\) to look up method definitions for raw method invocation \(e.m(\mathcal{S})\). Their definitions are similar to \(\text{mtype}\) and \(\text{mbody}\) except that, as in the rule below, when the definition of the method is found, \(\text{mbodyraw}\) checks whether the erasure changes the argument types \(U\) (i.e., whether they are all raw) and yields the erased argument and result types:

\[
\begin{align*}
\text{CT}(C) &= \text{class } C<\mathcal{X}<\mathcal{N}> <\mathcal{S} \mathcal{T}; \mathcal{K} \mathcal{R}> \quad \text{fields}(N) = \{U \mathcal{F}\} \\
\text{fields}(C) &= \{U, Y; C<\mathcal{T}>\}
\end{align*}
\]

3.4.2 Example: Under a class table including class Pair and List,

\[
\begin{align*}
\text{mtyperaw(setfst, Pair)} &= \{X<\text{Object, Object}, Y<\text{Object, Object}, \text{setfst, Pair}\} \\
&= \text{Object} \rightarrow \text{Pair} \quad \text{(with unchecked warning)} \\
\text{mbodyraw(makenule, List<T>)} &= (\ast, \text{new Null<*>()})
\end{align*}
\]

Besides, we introduce the new auxiliary definition \(\text{cargtype}\) to look up the argument types of a constructor; the definition is almost the same as \(\text{fields}\) except that it signals \(\text{unchecked warning}\) if erasure changes the types, as in the rule below.

\[
\begin{align*}
\text{CT}(C) &= \text{class } C<\mathcal{X}<\mathcal{N}> <\mathcal{S} \mathcal{T}; \mathcal{K} \mathcal{R}> \quad \text{cargtype}(N) = \{U \mathcal{F}\} \\
\text{cargtype}(C) &= \{U, Y; C<\mathcal{T}>\}
\end{align*}
\]

\text{unchecked warning if } |U, Y|_{\mathcal{X}<\mathcal{N}} \neq |U, Y|_{\mathcal{X}<\mathcal{N}} \mathcal{F}
Figure 4: Raw FGJ: Auxiliary Definitions
Subtyping, Well-Formed Types, and Typing

There are three additional subtyping rules for the bottom type and raw types: the bottom type is subtype of any type (S-Bot); a raw type \( C \) is subtype of \( D \) if \( D \) is the name of a superclass of \( C \) (S-Raw); and a cooked type \( C<:T > \) is subtype of the corresponding raw type \( C \) (S-Cooked/Raw). Moreover, we need to add a rule for subtyping derived from subclassing:

\[
\Delta \vdash C<:T > <: D<:*> \quad (S-RawSuper)
\]

As we already discussed, when the superclass is raw \( D \), parameterized types from the subclass are subtype of \( D<:*> \) rather than \( D \).

The judgment of unsafe subtyping is written \( \Delta \vdash S <:?:T \); it is used in comparing the types of formal arguments (of a constructor or method) with those of actual arguments. Unsafe subtyping includes safe subtyping (SU-Safe) and allows \( C \) to be unsafe subtype of the cooked type \( D<:*> \) (with unchecked warning) if a raw type \( C \) is safe subtype of another raw type \( D \) (SU-Unsafe).

The definition of type well-formedness is extended in a fairly straightforward manner: the bottom type is always well formed (WF-Bot) and a raw type is well formed if it is in the domain of \( CT (C) = \text{class } C<:N > <:N \{ \ldots \} \) (WF-Raw).

The typing rules are also mostly straightforward. When arguments are passed to a method constructor, their types can be unsafe subtype of the types expected. So, the rules for method invocation are as follows:

\[
\Delta; \Gamma \vdash e_0 \in T_0 \\
\Delta \vdash V \text{ ok} \\
\Delta \vdash V <: [\overline{V:T};P] \\
\Delta; \Gamma \vdash \overline{S} \in S \\
\Delta \vdash \overline{S} <: [\overline{V:T};P] \\
\Delta; \Gamma \vdash e_0.m<\overline{V}>(\overline{S}) \in [\overline{V:T};P] \\
\]

(T-Invk)
\[ \Delta; \Gamma \vdash e_0 \in T_0 \quad \text{mtyperaw}(m, |T_0|_\Delta) = \overline{C} \rightarrow C \quad \Delta; \Gamma \vdash \overline{\mathbb{S}} \in \overline{\mathbb{S}} \quad \Delta \vdash \overline{\mathbb{S}} \triangleleft \overline{C} \]  

\text{(T-INVKR)}

The rules T-NEW and T-NEWR for constructor invocation has to check, using cargtype, whether the constructor argument types are changed by erasure; similarly for invocation of super(0) in the rule T-CLASS. As we discussed above, the rule T-NEWR gives raw constructor invocation cooked type where all type variables are instantiated by *.

\[ \Delta \vdash C<\overline{U} > \ ok \quad \text{cargtype}(C<\overline{U}>) = T \ T \quad \Delta; \Gamma \vdash \overline{\mathbb{S}} \in \overline{\mathbb{S}} \quad \Delta \vdash \overline{\mathbb{S}} \triangleleft \overline{T} \]  

\text{(T-New)}

\[ \Delta \vdash C<\overline{S} > \ ok \quad \text{cargtype}(C) = T \ T \quad \Delta; \Gamma \vdash \overline{\mathbb{S}} \in \overline{\mathbb{S}} \quad \Delta \vdash \overline{\mathbb{S}} \triangleleft \overline{T} \]  

\text{(T-NEWR)}

In addition to the rules above, we require three rules relevant to the bottom type; we show two of them T-BOT-FIELD and T-BOT-INVK as representatives.

\[ \Delta; \Gamma \vdash e_0 \in * \quad \text{stupid warning} \]  

\text{(T-BOT-FIELD)}

\[ \Delta; \Gamma \vdash e_0 . f_1 \in * \]  

\[ \Delta; \Gamma \vdash e_0 . f \in * \]  

\text{(T-BOT-FIELD)}

\[ \Delta; \Gamma \vdash \overline{\mathbb{V}} \ ok \quad \Delta; \Gamma \vdash \overline{\mathbb{S}} \in \overline{\mathbb{S}} \quad \text{stupid warning} \]  

\text{(T-BOT-INVK)}

We allow any field access and any method invocation on an expression of the bottom type; the whole expression is also given the bottom type. Roughly speaking, the bottom type corresponds to the empty set; an expression of the bottom type never reduces to an object instantiation new N(\overline{\mathbb{S}}). Any operation on the element of the empty set is vacuously allowed and the result also belongs to the empty set. Similar rules can be found in an extension of System F\_2 with the bottom type [15]. As stupid casts, we signal stupid warning here as the GJ compiler actually rejects expressions of the bottom type; these rules are nevertheless needed to show type soundness through subject reduction.

Reduction Rules

Thanks to the auxiliary definitions, the computation rule for raw method invocation is very simple. (For congruence, we require two more rules, omitted from the figure.)

\[ \text{mbodyraw}(m, N) = (\overline{x}, e) \]  

\[ \text{new } N(\overline{\mathbb{S}}) . m(\overline{d}) \rightarrow [\overline{d}/\overline{x}, \text{new } N(\overline{\mathbb{S}})/\text{this}]e \]  

\text{(R-INVKR)}

3.5 Properties of Raw FGJ

Raw FGJ programs without unchecked warning enjoy the subject reduction and progress properties, which together guarantee type soundness.

3.5.1 Theorem [Raw FGJ subject reduction]: If \( \Delta; \Gamma \vdash e \in T \) without unchecked warning and \( e \rightarrow e' \), then \( \Delta; \Gamma \vdash e' \in T' \) without unchecked warning, for some \( T' \) such that \( \Delta \vdash T' \triangleleft T \).

3.5.2 Theorem [Raw FGJ progress]: Suppose \( e \) is a well-typed expression without unchecked warning.

1. If \( e \) includes new \( N_0(\overline{\mathbb{S}}) . f \) as a subexpression, then \( \text{fields}(N_0) = T \ T \) and \( f \in T \).

2. If \( e \) includes new \( N_0(\overline{\mathbb{S}}) . m<\overline{\mathbb{V}}>(\overline{\mathbb{D}}) \) as a subexpression, then \( \text{mbody}(m<\overline{\mathbb{V}}>, N_0) = (\overline{x}, e_0) \) and \( #(\overline{x}) = #(\overline{d}) \).

3. If \( e \) includes new \( N_0(\overline{\mathbb{S}}) . m(\overline{d}) \) as a subexpression, then \( \text{mbodyraw}(m, N_0) = (\overline{x}, e_0) \) and \( #(\overline{x}) = #(\overline{d}) \).

3.5.3 Theorem [Raw FGJ type soundness]: If \( \emptyset; \Gamma \vdash e \in T \) without unchecked warning and \( e \rightarrow^* e' \) with \( e' \) being a normal form, then \( e' \) is either (1) a value \( v \) with \( \emptyset; \Gamma \vdash v \in S \) and \( \emptyset \vdash S \triangleleft T \) or (2) an expression containing (P) new \( N(\overline{\mathbb{S}}) \) where \( \emptyset \vdash N \not\in P \).
### Expression Typing:

\[
\begin{align*}
\Delta; \Gamma \vdash x &\in \Gamma(x) & (T-VAR) \\
\Delta; \Gamma \vdash e_0 &\in T_0 & fields(bound_\Delta(T_0)) = T \ Y \\
\Delta; \Gamma \vdash e_0.f_i &\in T_i & (T-FIELD) \\
\Delta; \Gamma \vdash e_0 &\in T_0 & mtype(m, bound_\Delta(T_0)) = \langle \forall <P> U \rightarrow U \rangle \\
\Delta; \Gamma \vdash \forall &\in \mathbb{S} & \Delta; \Gamma \vdash \forall \vdash : \langle \forall \rangle \ U & (T-INVK) \\
\Delta; \Gamma \vdash e_0.m<\Delta> &\in \langle \forall \rangle \ U & (T-INVKR) \\
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & \Delta; \Gamma \vdash \forall &\in \mathbb{S} & \Delta; \Gamma \vdash e_0.m<\Delta> &\in \mathbb{S} & (T-BOT-INVK) \\
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & \Delta; \Gamma \vdash e_0.m<\Delta> &\in \mathbb{S} & (T-BOT-INVK-R) \\
\end{align*}
\]

### Method Typing:

\[
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & mtypeaw(m, T_0) = C \rightarrow C \\
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & \Delta; \Gamma \vdash e_0.m<\Delta> &\in \mathbb{S} & (T-NEW) \\
\Delta; \Gamma \vdash C<\mathbb{S}> &\text{ok} & \text{cargtype}(C<\mathbb{S}>) = T \ Y \\
\Delta; \Gamma \vdash C<\mathbb{S}> &\text{ok} & \text{cargtype}(C) = T \ Y \\
\Delta; \Gamma \vdash C<\mathbb{S}> &\text{ok} & \text{cargtype}(\mathbb{C}) = T \ Y \\
\Delta; \Gamma \vdash C<\mathbb{S}> &\text{ok} & \text{cargtype}(\mathbb{C}) = T \ Y \\
\Delta; \Gamma \vdash new C<\mathbb{S}> &\in \mathbb{S} & \mathbb{S} \vdash \forall \mathbb{U} &\text{ok} & \text{cargtype}(\mathbb{U}) = T \ Y \\
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & \Delta; \Gamma \vdash T_0 \vdash T_0 \vdash \mathbb{N} & (T-UCAST) \\
\Delta; \Gamma \vdash \Delta' \vdash T_0 \vdash \mathbb{N} & (T-DCAST) \\
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & \Delta; \Gamma \vdash e_0.m<\Delta> &\in \mathbb{S} & (T-SCAST) \\
\end{align*}
\]

### Class Typing:

\[
\begin{align*}
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & \Delta; \Gamma \vdash \Delta' \vdash \mathbb{N} & (T-METHOD) \\
\Delta; \Gamma \vdash e_0 &\in \mathbb{S} & \Delta; \Gamma \vdash \mathbb{N} &\text{ok} & \text{cargtype}(\mathbb{N}) = \mathbb{U} \ \mathbb{T} \ \mathbb{N} \ \mathbb{OK} & (T-CLASS) \\
\end{align*}
\]

---

### Raw FGJ: Typing Rules

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression typing:</td>
</tr>
<tr>
<td>Method typing:</td>
</tr>
<tr>
<td>Class typing:</td>
</tr>
</tbody>
</table>

---

### Raw FGJ: Reduction Rules

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-FIELD</td>
</tr>
<tr>
<td>R-INVK</td>
</tr>
<tr>
<td>R-INVKR</td>
</tr>
<tr>
<td>R-CAST</td>
</tr>
</tbody>
</table>

---

**Figure 6:** Raw FGJ: Typing Rules

**Figure 7:** Raw FGJ: Reduction Rules
4 Related Work

Pizza [14], another extension of Java with parameterized classes and the predecessor of GJ, makes some use of class names without type arguments and, in fact, existential types are used in the type system (only internally). They seem to be motivated to complement the restriction on typecasts due to lack of run-time type argument information. In Pizza (also in GJ, FGJ and Raw FGJ), downcasts are restricted so that the result will be the same whether programs are executed with or without run-time type arguments. Thus, for example, downcast from Object to Pair<String, String> is prohibited because, at run-time, the element type information is lost under the erased execution. In Pizza, one can write (Pair)e even though the notation Pair itself does not make sense as a type; it checks whether e is subtype of Pair<X, Y> for some X and Y and, thus, the whole expression is given the existential type ∃X, Y.Pair<X, Y>. In GJ, the same notation is treated as a raw type, which can appear everywhere a type is expected; of course, downcast to a raw type is allowed. In fact, this kind of use of raw types is deemed very important as well as the use for program evolution.

Other extensions of Java with polymorphic classes and methods [1, 13, 7, 17] did not provide functionality like raw types. According to the direct semantics of raw types shown here, it may be possible to augment them with raw types. In particular, our direct semantics would be directly applicable to extensions implementing type-passing semantics at either the source level [17] or the virtual machine level [13].

5 Discussions

We have formalized a novel feature, raw types, of GJ on top of a small core calculus Featherweight GJ. From the type-theoretic point of view, raw types are close to existential types except several unsafe operations such as coercion from a raw type to a cooked type are allowed. We have developed a type system and direct reduction semantics of raw types. One difficulty arose from the fact that type arguments for polymorphic method invocation may be missing from the expression. We have solved that problem by introducing the bottom type, which is used for any missing type argument. The typing rules are proved to be sound with respect to the operational semantics. In the course of the work, we have discovered a serious flaw in the type system given in the original specification. We are now developing a more formal connection between raw types and existential types, by showing encoding Raw FGJ to FGJ with existential types; such encoding may be useful to analyze the border between checked and unchecked programs less conservatively. To some extent, the encoding has turned out to be fairly straightforward.

Although it proves a sensible step towards understanding raw types, this work is rather preliminary; as well as checked programs, we also need to develop formal accounts of unchecked programs to investigate how raw types help program evolution. For example, it would be easily shown that an unchecked program is compiled to a well-typed FJ program (FGJ program that does not use any polymorphic features), like a similar result shown for FGJ [10]; thus, even though unchecked programs are not guaranteed to be statically safe, only failures will be due to stuck downcasts (in the FJ-level execution) Another important property expected is the evolution theorem, which can be stated roughly as follows: if a monomorphic class is replaced with an “appropriate” polymorphic version, the whole program that uses the evolved class can remain well typed (possibly with unchecked warnings). With a language with a property like that, it is easier to evolve programs: when one class is evolved to a polymorphic version, one need not modify other classes depending on the evolved class at the same time.

Acknowledgments

This work was supported by the University of Pennsylvania and the National Science Foundation under grant CCR-9701826, Principled Foundations for Programming with Objects. A part of this work was done while Igarashi was visiting University of Pennsylvania as a research fellow of the Japan Society of the Promotion of Science. We thank Martin Odersky for proposing the repair of the flaw we have found in the type system and Naoki Kobayashi for insightful comments.
References


