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 with 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`:

```
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:

```
class Pair<X,Y> {
  X fst; Y snd;
  Pair(X fst, Y snd){ this.fst=fst; this.snd=snd; }
  Pair<X,Y> setfst(X newfst){ fst=newfst; }
}
```

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 $\texttt{C<X}_1\texttt{,}\ldots\texttt{,X}_n\texttt{>}$ 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 a 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 $\texttt{e.m(}\overline{\texttt{e}}\texttt{)}$ where $\texttt{e}$ is of raw type, say $\texttt{C}$, is well typed without type arguments. At run-time, however, $\texttt{e}$ can reduce to an object $\texttt{new C<T>(}\overline{\texttt{d}}\texttt{)}$ 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 $\texttt{A}$, $\texttt{B}$, $\texttt{C}$, $\texttt{D}$, and $\texttt{E}$ range over class names; $\texttt{S}$, $\texttt{T}$, $\texttt{U}$, and $\texttt{V}$ range over types; $\texttt{X}$, $\texttt{Y}$, and $\texttt{Z}$ range type variables; $\texttt{N}$, $\texttt{P}$, and $\texttt{Q}$ range over non-variable types; $\texttt{L}$ ranges over class declarations; $\texttt{K}$ ranges over constructor declarations; and $\texttt{M}$ ranges over method declarations; $\texttt{f}$ and $\texttt{g}$ range over field names; $\texttt{m}$ ranges over method names; $\texttt{x}$ ranges over variables; and $\texttt{e}$ and $\texttt{d}$ range over expressions. We write $\overline{\texttt{f}}$ as shorthand for a possibly empty sequence $\texttt{f}_1\texttt{,}\ldots\texttt{,f}_n$ (and similarly for $\overline{\texttt{C}}$, $\overline{\texttt{x}}$, $\overline{\texttt{e}}$, etc.) and write $\overline{\texttt{M}}$ as shorthand for $\texttt{M}_1\ldots\texttt{M}_n$ (with no commas). We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. The length of a sequence $\overline{\texttt{x}}$ (or $\overline{\texttt{X}}$) is written $\#(\overline{\texttt{x}})$ (or $\#(\overline{\texttt{X}})$, respectively). We abbreviate operations on pairs of sequences in the obvious way, writing "$\overline{\texttt{C}}\ \overline{\texttt{f}}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1\texttt{,}\ldots\texttt{,C}_n\ \texttt{f}_n$" and "$\overline{\texttt{C}}\ \overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{C}_1\ \texttt{f}_1\texttt{;}\ldots\texttt{C}_n\ \texttt{f}_n\texttt{;}$" and "$\texttt{this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{;}$" as shorthand for "$\texttt{this.f}_1\texttt{=f}_1\texttt{;}\ldots\texttt{this.f}_n\texttt{=f}_n\texttt{;}$" and "$\texttt{<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}$" as shorthand for "$\texttt{<X}_1\triangleleft\texttt{N}_1\texttt{, }\ldots\texttt{, X}_n\triangleleft\texttt{N}_n\texttt{>}$. Sequences of field declarations, parameter names, type variables, and method declarations are assumed to contain no duplicate names. As in GJ, the empty brackets $\texttt{<>}$ are omitted.

A class declaration consists of its name ($\texttt{class C}$), type parameters ($\overline{\texttt{X}}$) with their bounds ($\overline{\texttt{N}}$), fields ($\overline{\texttt{T}}\ \overline{\texttt{f}}$), one constructor ($\texttt{K}$), and methods ($\overline{\texttt{M}}$); moreover, every class must explicitly declare its supertype $\texttt{N}$ with $\triangleleft$ (abbreviation of $\texttt{extends}$) even if it is $\texttt{Object}$. 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:

$\texttt{T}(\texttt{S}, \texttt{U}, \texttt{V})$ (types)

$::=$	$\texttt{X}$	type variables
$\mid$	$\texttt{N}$	non-variable types

$\texttt{N}(\texttt{P}, \texttt{Q})$

$::= \quad \texttt{C<}\overline{\texttt{T}}\texttt{>}$

$\texttt{L}$ (class declarations)

$::= \quad \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \; \{\overline{\texttt{T}} \; \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\}$

$\texttt{K}$ (constructors)

$::= \quad \texttt{C(}\overline{\texttt{T}} \; \overline{\texttt{f}}\texttt{)\{ super(}\overline{\texttt{f}}\texttt{); this.}\overline{\texttt{f}}\texttt{=}\overline{\texttt{f}}\texttt{; \}}$

$\texttt{M}$ (methods)

$::= \quad \texttt{<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>T m (}\overline{\texttt{T}} \; \overline{\texttt{x}}\texttt{)\{ return e; \}}$

$\texttt{e}$ (expressions)

$::=$	$\texttt{x}$	variables, $\texttt{this}$
$\mid$	$\texttt{e.f}$	field access
$\mid$	$\texttt{e.m<}\overline{\texttt{T}}\texttt{>(}\overline{\texttt{e}}\texttt{)}$	method invocation
$\mid$	$\texttt{new N(}\overline{\texttt{e}}\texttt{)}$	object instantiation
$\mid$	$\texttt{(N)e}$	typecast

Bound of type:

$$\begin{aligned} \mathit{bound}_\Delta(\texttt{X}) &= \Delta(\texttt{X}) \\ \mathit{bound}_\Delta(\texttt{N}) &= \texttt{N} \end{aligned}$$

Subclassing:

$$\texttt{C} \trianglelefteq \texttt{C} \qquad \frac{\texttt{C} \trianglelefteq \texttt{D} \quad \texttt{D} \trianglelefteq \texttt{E}}{\texttt{C} \trianglelefteq \texttt{E}}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{T}}\texttt{>} \; \{\ldots\}}{\texttt{C} \trianglelefteq \texttt{D}}$$

Field lookup:

$$\mathit{fields}(\texttt{Object}) = \bullet$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \; \{\overline{\texttt{S}} \; \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \qquad \mathit{fields}([\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N}) = \overline{\texttt{U}} \; \overline{\texttt{g}}}{\mathit{fields}(\texttt{C<}\overline{\texttt{T}}\texttt{>}) = \overline{\texttt{U}} \; \overline{\texttt{g}}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{S}} \; \overline{\texttt{f}}}$$

Method type lookup:

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \; \{\overline{\texttt{S}} \; \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \qquad \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}} \; \overline{\texttt{x}}\texttt{) \{return e;\}} \in \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = [\overline{\texttt{T}}/\overline{\texttt{X}}](\texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \to \texttt{U})}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \; \{\overline{\texttt{S}} \; \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mtype}(\texttt{m}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mtype}(\texttt{m}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})}$$

Method body lookup:

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \; \{\overline{\texttt{S}} \; \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \qquad \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{> U m (}\overline{\texttt{U}} \; \overline{\texttt{x}}\texttt{) \{return } \texttt{e}_0\texttt{;\}} \in \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = (\overline{\texttt{x}}, [\overline{\texttt{T}}/\overline{\texttt{X}}, \overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{e}_0)}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \; \{\overline{\texttt{S}} \; \overline{\texttt{f}}\texttt{; K } \overline{\texttt{M}}\} \qquad \texttt{m} \notin \overline{\texttt{M}}}{\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{C<}\overline{\texttt{T}}\texttt{>}) = \mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})}$$

Valid downcast:

$$\frac{\mathit{dcast}(\texttt{S}, \texttt{T}) \qquad \mathit{dcast}(\texttt{T}, \texttt{U})}{\mathit{dcast}(\texttt{S}, \texttt{U})}$$

$$\frac{\mathit{CT}(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N} \; \{\ldots\} \qquad \overline{\texttt{X}} = \mathit{FV}(\texttt{N})}{\mathit{dcast}(\texttt{C<}\overline{\texttt{T}}\texttt{>}, [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{N})}$$

($\mathit{FV}(\texttt{N})$ denotes the type variable in $\texttt{N}$.)

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 fields of the class. As in Java and GJ, fields inherited from superclasses are initialized by `super(`$\overline{\texttt{f}}$`);` and newly declared fields by `this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;`, 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 $\overline{\texttt{T}}$ for method invocation `e.m<`$\overline{\texttt{T}}$`>(`$\overline{\texttt{e}}$`)`. 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 name.

A class table CT is a mapping from class names `C` to class declarations `L`; a *program* is a pair $(CT, \texttt{e})$ 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) $CT(\texttt{C}) = \texttt{class C...}$ for every $\texttt{C} \in dom(CT)$; (2) $\texttt{Object} \notin dom(CT)$; (3) for every class name `C` (except `Object`) appearing anywhere in CT, we have $\texttt{C} \in dom(CT)$; and (4) there are no cycles in the reflexive and transitive closure of the relation between class names induced by $\triangleleft$ clauses in CT. (The formal definition of that relation $\trianglelefteq$ appears in Figure 1.) By the condition (1), we can identify a class table with a sequence of class declarations in the obvious way.

For the typing and reduction rules, we need a few auxiliary definitions, given also in Figure 1. A type environment Δ is a finite mapping from type variables to nonvariable types, written $\overline{\texttt{X}}$`<:`$\overline{\texttt{N}}$, that takes each type variable to its bound. We write $bound_\Delta(\texttt{T})$ for the upper bound of `T` in Δ. The fields of a nonvariable type `N`, written *fields*(`N`), are a sequence of corresponding types and field names, $\overline{\texttt{T}}\ \overline{\texttt{f}}$. The type of the method invocation `m` at nonvariable type `N`, written *mtype*(`m`, `N`), is a type of the form $\texttt{<}\overline{\texttt{X}}\triangleleft\overline{\texttt{N}}\texttt{>}\overline{\texttt{U}}\rightarrow\texttt{U}_0$, where $\overline{\texttt{X}}$ are type parameters, $\overline{\texttt{U}}$ are argument types, and $\texttt{U}_0$ is the result type. In this form, the variables $\overline{\texttt{X}}$ are bound in $\overline{\texttt{N}}$, $\overline{\texttt{U}}$, and $\texttt{U}_0$ and we regard α-convertible ones as equivalent; application of type substitution $[\overline{\texttt{T}}/\overline{\texttt{X}}]$ is defined in the customary manner. The body of the method invocation `m` at nonvariable type `N` with type parameters $\overline{\texttt{V}}$, written *mbody*(`m<`$\overline{\texttt{V}}$`>`, `N`), is a pair, written $(\overline{\texttt{x}},\texttt{e})$, of a sequence of parameters $\overline{\texttt{x}}$ and an expression `e`. (Note that the functions *mtype*(`m`, `N`) and *mbody*(`m<`$\overline{\texttt{V}}$`>`, `N`) are both partial functions: since `Object` is assumed to have no methods in FGJ, both *mtype*(`m`, `Object`) and *mbody*(`m<`$\overline{\texttt{V}}$`>`, `Object`) are undefined.) We write $\texttt{m} \notin \overline{\texttt{M}}$ to mean the method definition of the name `m` is not included in $\overline{\texttt{M}}$.

2.2 Type System

The type system of FGJ consists of three forms of judgments: one for subtyping $\Delta \vdash \texttt{S} \texttt{ <: } \texttt{T}$, one for type well-formedness $\Delta \vdash \texttt{T}$ ok, and one for typing $\Delta; \Gamma \vdash \texttt{e} \in \texttt{T}$, where Γ is an environment, a finite mapping from variables to types, written $\overline{\texttt{x}}\texttt{:}\overline{\texttt{T}}$. Figure 2 shows the rules to derive these judgments. We abbreviate a sequence of judgments, writing $\Gamma \vdash \overline{\texttt{S}} \texttt{ <: } \overline{\texttt{T}}$ as shorthand for $\Gamma \vdash \texttt{S}_1 \texttt{ <: } \texttt{T}_1, \ldots, \Gamma \vdash \texttt{S}_n \texttt{ <: } \texttt{T}_n$ and $\Gamma \vdash \overline{\texttt{T}}$ ok as shorthand for $\Gamma \vdash \texttt{T}_1$ ok$, \ldots, \Gamma \vdash \texttt{T}_n$ ok and $\Delta; \Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{T}}$ as shorthand for $\Delta; \Gamma \vdash \texttt{e}_1 \in \texttt{T}_1, \ldots, \Delta; \Gamma \vdash \texttt{e}_n \in \texttt{T}_n$.

Subtyping is the reflexive and transitive closure of the relation induced by $\triangleleft$ clauses. Type parameters are *invariant* with regard to subtyping (for the usual reasons: type parameters can be used both positively and negatively), so $\Delta \vdash \overline{\texttt{T}} \texttt{ <: } \overline{\texttt{U}}$ does *not* imply $\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \texttt{ <: } \texttt{C<}\overline{\texttt{U}}\texttt{>}$.

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

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:

$$\Delta \vdash \mathtt{T} \mathrel{<:} \mathtt{T} \qquad \frac{\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T} \qquad \Delta \vdash \mathtt{T} \mathrel{<:} \mathtt{U}}{\Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{U}}$$

$$\Delta \vdash \mathtt{X} \mathrel{<:} \Delta(\mathtt{X}) \qquad \frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\{\ldots\}}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}}$$

Well-formed types:

$$\Delta \vdash \mathtt{Object}\ \text{ok} \qquad \frac{\mathtt{X} \in dom(\Delta)}{\Delta \vdash \mathtt{X}\ \text{ok}}$$

$$\frac{\begin{array}{c} CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{\ldots\}} \\ \Delta \vdash \overline{\mathtt{T}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{T}} \mathrel{<:} [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{N}} \end{array}}{\Delta \vdash \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}\ \text{ok}}$$

Expression typing:

$$\Delta; \Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x})$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad fields(bound_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.f}_i \in \mathtt{T}_i}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ mtype(\mathtt{m}, bound_\Delta(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{V}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \\ \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}} \end{array}}{\Delta; \Gamma \vdash \mathtt{e}_0\mathtt{.m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{e}}\mathtt{)} \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}}$$

$$\frac{\begin{array}{c} \Delta \vdash \mathtt{N}\ \text{ok} \qquad fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta; \Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:} \overline{\mathtt{T}} \end{array}}{\Delta; \Gamma \vdash \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)} \in \mathtt{N}}$$

$$\frac{\Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 \mathrel{<:} \mathtt{N}}{\Delta; \Gamma \vdash \mathtt{(N)}\mathtt{e}_0 \in \mathtt{N}}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{N}\ \text{ok} \\ \Delta \vdash \mathtt{N} \mathrel{<:} bound_\Delta(\mathtt{T}_0) \qquad dcast(\mathtt{N}, bound_\Delta(\mathtt{T}_0)) \end{array}}{\Delta; \Gamma \vdash \mathtt{(N)}\mathtt{e}_0 \in \mathtt{N}}$$

$$\frac{\begin{array}{c} \Delta; \Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad bound_\Delta(\mathtt{T}_0) = \mathtt{D<}\overline{\mathtt{U}}\mathtt{>} \qquad \mathtt{N} = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \\ \Delta \vdash \mathtt{N}\ \text{ok} \qquad \mathtt{C} \not\trianglelefteq \mathtt{D} \qquad \mathtt{D} \not\trianglelefteq \mathtt{C} \qquad \textit{stupid warning} \end{array}}{\Delta; \Gamma \vdash \mathtt{(N)}\mathtt{e}_0 \in \mathtt{N}}$$

Method typing:

$$\frac{\begin{array}{c} mtype(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Z}} \lhd \overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}_0 \text{ implies} \\ \overline{\mathtt{P}}\mathtt{,}\overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{Q}}\mathtt{,}\overline{\mathtt{U}}) \text{ and } \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \vdash \mathtt{T}_0 \mathrel{<:} [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U}_0 \end{array}}{override(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T}_0)}$$

$$\frac{\begin{array}{c} \Delta = \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\mathtt{<:}\overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \text{ok} \\ \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 \in \mathtt{S} \qquad \Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T} \\ CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{\ldots\}} \\ override(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T}) \end{array}}{\mathtt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\mathtt{>\ T\ m\ (}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\ \{return\ e}_0\mathtt{;\}\ OK\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>}}$$

Class typing:

$$\frac{\begin{array}{c} \overline{\mathtt{X}}\mathtt{<:}\overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \mathtt{N}, \overline{\mathtt{T}}\ \text{ok} \\ fields(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \\ \mathtt{K} = \mathtt{C(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,\ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{\ =\ }\overline{\mathtt{f}}\mathtt{;\}} \end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>} \lhd \mathtt{N\ \{}\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K\ }\overline{\mathtt{M}}\mathtt{\}\ OK}}$$

Computation:

$$\frac{fields(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i}$$

$$\frac{mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)} \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0}$$

$$\frac{\emptyset \vdash \mathtt{N} \mathrel{<:} \mathtt{P}}{\mathtt{(P)(new\ N(}\overline{\mathtt{e}}\mathtt{))} \longrightarrow \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)}}$$

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 *dcast*(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 $(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>})\mathtt{e}$ is allowed only if all type arguments $\overline{\mathtt{T}}$ 'contribute' to decide $\Delta \vdash \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>} \texttt{<:}\ \mathit{bound}_\Delta(\mathtt{T})$ where T is the type of e—the condition is expressed by $\overline{\mathtt{X}} = \mathtt{FV(N)}$ in the definition of *dcast*(S, T). For example, `(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 $\mathtt{M\ OK\ IN\ C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}$, read "method declaration M is ok if it occurs in class $\mathtt{C}\texttt{<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\texttt{>}$." 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 $\mathtt{C}\texttt{<}\overline{\mathtt{X}}\texttt{>}$. (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 α-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 OK`, 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 $\mathtt{e} \longrightarrow \mathtt{e}'$, read "expression e reduces to expression $\mathtt{e}'$ in one step." We write $\longrightarrow^*$ for the reflexive and transitive closure of $\longrightarrow$. 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 $\mathtt{new\ N}(\overline{\mathtt{e}}).\mathtt{f}_i$ looks up and obtains field names $\overline{\mathtt{f}}$ of N with *fields*(N); then it reduces to the constructor argument $\mathtt{e}_i$ of the corresponding position. Method invocation $\mathtt{new\ N}(\overline{\mathtt{e}}).\mathtt{m}(\overline{\mathtt{d}})$ first looks up *mbody*(m, N) and obtains a pair of a sequence of formal arguments $\overline{\mathtt{x}}$ and the method body; then, it reduces to the method body in which $\overline{\mathtt{x}}$ are replaced with the actual arguments $\overline{\mathtt{d}}$ and `this` with the receiver $\mathtt{new\ N}(\overline{\mathtt{e}})$. We write $[\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{e}/\mathtt{y}]\mathtt{e}_0$ to stand for replacing $\mathtt{x}_1$ by $\mathtt{d}_1, \ldots, \mathtt{x}_n$ by $\mathtt{d}_n$, and y by e in the expression $\mathtt{e}_0$. Typecast $(\mathtt{P})\mathtt{new\ N}(\overline{\mathtt{e}})$ 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 $\mathtt{e} \longrightarrow \mathtt{e}'$ then $\mathtt{e.f} \longrightarrow \mathtt{e}'\mathtt{.f}$, 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 Pair<A,B>(new A(), new B()).setfst(new B()).snd` evaluates to `new B()` via the following reduction steps (where the next subexpression to be reduced is underlined at each step):

$$
\begin{array}{ll}
& \underline{\texttt{new Pair<A,B>(new A(), new B()).setfst(new B())}}\texttt{.snd} \\
\longrightarrow & \left[\begin{array}{l}\texttt{A/X, B/Y, new B()/newfst,}\\ \texttt{new Pair<A,B>(new A(), new B())/this}\end{array}\right]\texttt{new Pair<X,Y>(newfst, this.snd)}\\
= & \texttt{new Pair<A,B>(new B(), }\underline{\texttt{new Pair<A,B>(new A(), new B()).snd}}\texttt{).snd}\\
\longrightarrow & \underline{\texttt{new Pair<A,B>(new B(), new B()).snd}}\\
\longrightarrow & \texttt{new B()}
\end{array}
$$

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 $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$ and $\texttt{e} \longrightarrow \texttt{e}'$, then $\Delta;\Gamma \vdash \texttt{e}' \in \texttt{T}'$, for some $\texttt{T}'$ such that $\Delta \vdash \texttt{T}' \texttt{ <: } \texttt{T}$.

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

(1) If e includes $\texttt{new N}_0\texttt{(}\overline{\texttt{e}}\texttt{).f}$ as a subexpression, then $\mathit{fields}(\texttt{N}_0) = \overline{\texttt{T}}\ \overline{\texttt{f}}$ and $\texttt{f} \in \overline{\texttt{f}}$.

(2) If e includes $\texttt{new N}_0\texttt{(}\overline{\texttt{e}}\texttt{).m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)}$ as a subexpression, then $\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}_0) = (\overline{\texttt{x}}, \texttt{e}_0)$ and $\#(\overline{\texttt{x}}) = \#(\overline{\texttt{d}})$.

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

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 `Pair<X,Y>` from the last section and the following class `Client`, rewritten in Raw FGJ, from the introduction.

```
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 `fst` and `snd` are given type `Object`, which is the bound of `X` and `Y`, and the constructor takes a pair of `Objects`. The method `setfst` (from `Pair`) is given type `Object`→`Pair`, in which type parameters are removed from the result type `Pair<X,Y>`. Concerning subtyping, any parameterized type `Pair<S,T>` is subtype of the raw type `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 $\texttt{C<}\overline{\texttt{T}}\texttt{>}$. For example, passing an expression of raw type `Pair` to a method expecting an argument of `Pair<String,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 `setfst` on `Pair` is unchecked because the argument type is changed by erasure from `X` to `Object`. Similarly, `new Pair(`e_1`,`e_2`)` (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 `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 `Pair<X,Y>` and `Client` together are unchecked because of the invocation of `setfst` in `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 `setfst` is unchecked.

```
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 *some* implementation of a polymorphic class of the same name `C`. By this analogy, for example, the raw type `Pair` would be considered the existential type `∃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 `Object`; similarly `p.setfst(x)` would be given type `Pair<X,Y>`, promoted to the raw type `Pair`. Actually, coercion from a parameterized type to a raw type would be considered existential packing. For example, passing an expression `e` of type `Pair<A,B>` to where `Pair` is expected roughly corresponds to

```
pack [X=A,Y=B;e] as ∃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 `∃X<:Object,Y<:Object.Pair<X,Y>` again; the method `setfst` would be given type `X→Pair<X,Y>` but, since we do not know the identity of `X`, we can neither approximate `X` by `Object` nor use the method in a type-safe manner. In fact, with the following subclass `IdPair` of `Pair`

```
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

```
new Client().setfst(new IdPair(new Id(), new Id()), new Object())
```

fails at the method invocation `newfst.id()`.

A similar argument applies to raw constructor invocation: for example, with the following class:

```
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; }
}
```

the expression

```
new D(new Pair<Object,Id>(new Object(), new Id())).loophole()
```

fails at the method invocation `this.g.fst.id`, too. Notice that the argument type of the raw constructor `D` is changed from `Pair<Id,Id>` to `Pair` by erasure, thus accepting an object of `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.[1]

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 `List` implementing a polymorphic method to obtain a null list for an arbitrary element type:

```
class List<X extends Object> extends Object {
  ...
  <Y extends Object> List<Y> makenull() { return new Null<Y>(); }
}
class Null<X extends Object> extends List<X> { ... }
```

Since the type system erases even type parameters of methods, method invocation `e.makenull()`, where `e` is of raw type `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, `e` may eventually reduce to an object `new List<T>(`$\overline{\texttt{d}}$`)` of parameterized type; then, it is required to 'make up' appropriate type arguments for the type parameter `Y` of `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, `new List<T>(`$\overline{\texttt{d}}$`).makenull()` reduces to `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.

[1]The dummy type parameter `X` is required when readers try this example with the GJ compiler: the GJ compiler would identify the type expression `D` with the parameterized type `D<>` when the class `D` is declared without type parameters. Declaring `D` with type parameters makes sure that `D` is raw.

T(S, U, V) (types)		
::=	X	type variables
|	N	non-variable types
|	*	bottom type
N(P, Q)		
::=	$\mathtt{C<\overline{T}>}$	cooked types
|	C	raw types
L (class declarations)		
::=	$\mathtt{class\ C<\overline{X} \triangleleft \overline{N}> \triangleleft N\ \{\overline{T}\ \overline{f};\ K\ \overline{M}\}}$	

K (constructors)		
::=	$\mathtt{C(\overline{T}\ \overline{f})\{\ super(\overline{f});\ this.\overline{f}=\overline{f};\ \}}$	
M (methods)		
::=	$\mathtt{<\overline{X} \triangleleft \overline{N}>\ T\ m(\overline{T}\ \overline{x})\{\ return\ e;\ \}}$	
e (expressions)		
::=	x	variables, this
|	e.f	field access
|	$\mathtt{e.m<\overline{T}>(\overline{e})}$	cooked method invocation
|	$\mathtt{e.m(\overline{e})}$	raw method invocation
|	$\mathtt{new\ N(\overline{e})}$	object instantiation
|	(N)e	typecasts

Figure 3: Raw FGJ: Syntax

- 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 `List`. Hence, returning an expression of parameterized type (for example, `Null<*>`, which is subtype of `List`) poses no problem.

Similarly to method invocation, instantiation of raw type, say $\mathtt{new\ Pair(e_1,e_2)}$, 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 $\mathtt{new\ Pair(e_1,e_2)}$ type `Pair<*,*>`, rather than `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 $\mathtt{D<\overline{X}>}$, any parameterized type $\mathtt{D<\overline{T}>}$ from the subclass is subtype of `C<*,...,*>` 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 $\mathtt{C<\overline{X}\ extends\ \overline{N}>}$ is typed under the assumption that `this` is given type $\mathtt{C<\overline{X}>}$ with $\mathtt{\overline{X}{<:}\overline{N}}$. Then, for invocation of the method to be type-preserving, `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 $\mathtt{C<\overline{X}>}$, the type of `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 *cooked*, which has type parameters, or *raw*; the bottom type `*` is included in the set of types. We often use $\overline{*}$ for a sequence of the bottom types `*`,...,`*` (without subscripts). Concerning expressions, method invocation $\mathtt{e.m(\overline{e})}$ 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 $(CT, \mathtt{e})$). 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 $\mathtt{class\ C<> \triangleleft N\ \ldots}$ to $\mathtt{class\ C \triangleleft N\ \ldots}$ and a method definition $\mathtt{<>T\ m(\overline{T}\ \overline{x})\ \ldots}$ to $\mathtt{T\ m(\overline{T}\ \overline{x})\ \ldots}$ for conciseness. (GJ is in favor of the cooked type `C<>` rather than the raw type `C` when you omit `<>`.)

We require the definition of *erasure* $|\mathtt{T}|_\Delta$ of type $\mathtt{T}$ under Δ, defined as the least raw supertype. Note that both $\mathit{bound}_\Delta(\mathtt{*})$ and $|\mathtt{*}|_\Delta$ are undefined.

$$\mathit{head}(\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = \mathit{head}(\mathtt{C}) = \mathtt{C}$$
$$|\mathtt{T}|_\Delta = \mathit{head}(\mathit{bound}_\Delta(\mathtt{T}))$$

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

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\texttt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathit{fields}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{\mathit{fields}(\mathtt{C}) = |\overline{\mathtt{U}}|_{\overline{\mathtt{X}\texttt{<:}\mathtt{N}}}\ \overline{\mathtt{g}},\ |\overline{\mathtt{S}}|_{\overline{\mathtt{X}\texttt{<:}\mathtt{N}}}\ \overline{\mathtt{f}}}$$

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

$$\begin{aligned}\mathit{fields}(\mathtt{Pair}) &= |\mathtt{X}|_{\mathtt{X<:Object,\ Y<:Object}}\ \mathtt{fst}, |\mathtt{Y}|_{\mathtt{X<:Object,\ Y<:Object}}\ \mathtt{snd}\\ &= \mathtt{Object\ fst,\ Object\ snd}\end{aligned}$$

Notice that types $\overline{\mathtt{S}}$ of the fields inherited from subclasses are erased under the type environment $\overline{\mathtt{X}\texttt{<:}\mathtt{N}}$ 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 *mtype* and *mbody*, we define (partial) functions *mtyperaw* and *mbodyraw* to look up method definitions for raw method invocation $\mathtt{e.m}(\overline{\mathtt{e}})$. Their definitions are similar to *mtype* and *mbody* except that, as in the rule below, when the definition of the method is found, *mtyperaw* checks whether the erasure changes the argument types $\overline{\mathtt{U}}$ (i.e., whether they are all raw) and yields the erased argument and result types:

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\texttt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \texttt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\texttt{>}\ \mathtt{U}_0\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e};\ \} \in \overline{\mathtt{M}}\\ \Delta' = \overline{\mathtt{X}}\texttt{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\texttt{<:}\overline{\mathtt{P}} \qquad \mathit{unchecked\ warning}\ \text{if}\ |\overline{\mathtt{U}}|_{\Delta'} \neq \overline{\mathtt{U}}\end{array}}{\mathit{mtyperaw}(\mathtt{m}, \mathtt{C}) = |\overline{\mathtt{U}}|_{\Delta'} \rightarrow |\mathtt{U}_0|_{\Delta'}}$$

The function *mbodyraw*(m, $\mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}$) can take a cooked type as the second argument because, as we discussed above, the receiver of raw method invocation may eventually be an instance of cooked type. The first rule below returns the method body where `*` is substituted for the type parameters that lacks the corresponding actual. The second rule calls the first with an appropriate number of the bottom types, replacing all the type variables with `*`.

$$\frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\texttt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \texttt{<}\overline{\mathtt{Y}} \lhd \overline{\mathtt{P}}\texttt{>}\ \mathtt{U}\ \mathtt{m}(\overline{\mathtt{U}}\ \overline{\mathtt{x}})\{\ \mathtt{return}\ \mathtt{e}_0;\ \} \in \overline{\mathtt{M}}\end{array}}{\mathit{mbodyraw}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{T}}\texttt{>}) = (\overline{\mathtt{x}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{e}_0)} \qquad \frac{\begin{array}{c}\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\texttt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\}\\ \#(\overline{\mathtt{X}}) = \#(\overline{\mathtt{*}})\end{array}}{\mathit{mbodyraw}(\mathtt{m}, \mathtt{C}) = \mathit{mbodyraw}(\mathtt{m}, \mathtt{C}\texttt{<}\overline{\mathtt{*}}\texttt{>})}$$

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

$$\begin{aligned}\mathit{mtyperaw}(\mathtt{setfst}, \mathtt{Pair}) &= |\mathtt{X}|_{\mathtt{X<:Object,\ Y<:Object}} \rightarrow |\mathtt{Pair\texttt{<}X,Y\texttt{>}}|_{\mathtt{X<:Object,\ Y<:Object}}\\ &= \mathtt{Object} \rightarrow \mathtt{Pair} \quad (\text{with } \mathit{unchecked\ warning})\\ \mathit{mbodyraw}(\mathtt{makenull}, \mathtt{List\texttt{<}T\texttt{>}}) &= (\bullet, \mathtt{new\ Null\texttt{<}*\texttt{>}()})\end{aligned}$$

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

$$\frac{\mathit{CT}(\mathtt{C}) = \mathtt{class}\ \mathtt{C}\texttt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\texttt{>} \lhd \mathtt{N}\ \{\overline{\mathtt{S}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\} \qquad \mathit{cargtype}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \mathit{unchecked\ warning}\ \text{if}\ |\overline{\mathtt{U}}, \overline{\mathtt{S}}|_{\overline{\mathtt{X}\texttt{<:}\mathtt{N}}} \neq \overline{\mathtt{U}}, \overline{\mathtt{S}}}{\mathit{cargtype}(\mathtt{C}) = |\overline{\mathtt{U}}|_{\overline{\mathtt{X}\texttt{<:}\mathtt{N}}}\ \overline{\mathtt{g}},\ |\overline{\mathtt{S}}|_{\overline{\mathtt{X}\texttt{<:}\mathtt{N}}}\ \overline{\mathtt{f}}}$$

Bound and erasure of type:

$$bound_\Delta(\mathtt{X}) = \Delta(\mathtt{X})$$
$$bound_\Delta(\mathtt{N}) = \mathtt{N}$$
$$head(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = head(\mathtt{C}) = \mathtt{C}$$
$$|\mathtt{T}|_\Delta = head(bound_\Delta(\mathtt{T}))$$

Subclassing:

$$\mathtt{C} \trianglelefteq \mathtt{C} \qquad \frac{\mathtt{C} \trianglelefteq \mathtt{D} \quad \mathtt{D} \trianglelefteq \mathtt{E}}{\mathtt{C} \trianglelefteq \mathtt{E}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{...\}}}{\mathtt{C} \trianglelefteq head(\mathtt{N})}$$

Field lookup:

$$fields(\mathtt{Object}) = \bullet$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad fields([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{fields(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad fields(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{fields(\mathtt{C}) = |\overline{\mathtt{U}}|_{\overline{\mathtt{X}}<:\overline{\mathtt{N}}}\ \overline{\mathtt{g}}, |\overline{\mathtt{S}}|_{\overline{\mathtt{X}}<:\overline{\mathtt{N}}}\ \overline{\mathtt{f}}}$$

Constructor type lookup:

$$cargtype(\mathtt{Object}) = \bullet$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad cargtype([\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}}{cargtype(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\overline{\mathtt{S}}\ \overline{\mathtt{f}}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad cargtype(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \textit{unchecked warning if } |\overline{\mathtt{U}}, \overline{\mathtt{S}}|_{\overline{\mathtt{X}}<:\overline{\mathtt{N}}} \neq \overline{\mathtt{U}}, \overline{\mathtt{S}}}{cargtype(\mathtt{C}) = |\overline{\mathtt{U}}|_{\overline{\mathtt{X}}<:\overline{\mathtt{N}}}\ \overline{\mathtt{g}}, |\overline{\mathtt{S}}|_{\overline{\mathtt{X}}<:\overline{\mathtt{N}}}\ \overline{\mathtt{f}}}$$

Method type lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>\ U\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = [\overline{\mathtt{T}}/\overline{\mathtt{X}}](\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mtype(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mtype(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>\ U_0\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e;\ \}} \in \overline{\mathtt{M}} \qquad \Delta' = \overline{\mathtt{X}}<:\overline{\mathtt{N}}, \overline{\mathtt{Y}}<:\overline{\mathtt{P}} \quad \textit{unchecked warning if } |\overline{\mathtt{U}}|_{\Delta'} \neq \overline{\mathtt{U}}}{mtyperaw(\mathtt{m}, \mathtt{C}) = |\overline{\mathtt{U}}|_{\Delta'} \rightarrow |\mathtt{U}_0|_{\Delta'}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{T}}\mathtt{>\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}} \quad mtype(\mathtt{m}, \mathtt{D<}\overline{\mathtt{T}}\mathtt{>}) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}_0 \qquad \Delta' = \overline{\mathtt{X}}<:\overline{\mathtt{N}}, \overline{\mathtt{Y}}<:\overline{\mathtt{P}} \quad \textit{unchecked warning if } |\overline{\mathtt{U}}|_{\Delta'} \neq \overline{\mathtt{U}}}{mtyperaw(\mathtt{m}, \mathtt{C}) = |\overline{\mathtt{U}}|_{\Delta'} \rightarrow |\mathtt{U}_0|_{\Delta'}}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mtyperaw(\mathtt{m}, \mathtt{C}) = mtyperaw(\mathtt{m}, \mathtt{D})}$$

Method body lookup:

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>\ U\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0;\ \}} \in \overline{\mathtt{M}}}{mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = (\overline{\mathtt{x}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{e}_0)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mbody(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>\ U\ m(}\overline{\mathtt{U}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e_0;\ \}} \in \overline{\mathtt{M}}}{mbodyraw(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = (\overline{\mathtt{x}}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}, \overline{\mathtt{*}}/\overline{\mathtt{Y}}]\mathtt{e}_0)}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \mathtt{m} \notin \overline{\mathtt{M}}}{mbodyraw(\mathtt{m}, \mathtt{C<}\overline{\mathtt{T}}\mathtt{>}) = mbodyraw(\mathtt{m}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{}\overline{\mathtt{S}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\mathtt{\}} \qquad \#(\overline{\mathtt{X}}) = \#(\overline{\mathtt{*}})}{mbodyraw(\mathtt{m}, \mathtt{C}) = mbodyraw(\mathtt{m}, \mathtt{C<}\overline{\mathtt{*}}\mathtt{>})}$$

Valid downcast:

$$\frac{dcast(\mathtt{S}, \mathtt{T}) \qquad dcast(\mathtt{T}, \mathtt{U})}{dcast(\mathtt{S}, \mathtt{U})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N\ \{...\}} \qquad \overline{\mathtt{X}} = FV(\mathtt{N})}{dcast(\mathtt{C<}\overline{\mathtt{T}}\mathtt{>}, [\overline{\mathtt{T}}/\overline{\mathtt{X}}]\mathtt{N})}$$

$$\frac{CT(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{D<}\overline{\mathtt{T}}\mathtt{>\ \{...\}}}{dcast(\mathtt{C}, \mathtt{D})}$$

Figure 4: Raw FGJ: Auxiliary Definitions

Subtyping:

$$\Delta \vdash \texttt{T} \mathrel{\texttt{<:}} \texttt{T} \qquad \text{(S-Refl)}$$

$$\frac{\Delta \vdash \texttt{S} \mathrel{\texttt{<:}} \texttt{T} \qquad \Delta \vdash \texttt{T} \mathrel{\texttt{<:}} \texttt{U}}{\Delta \vdash \texttt{S} \mathrel{\texttt{<:}} \texttt{U}} \qquad \text{(S-Trans)}$$

$$\Delta \vdash \texttt{X} \mathrel{\texttt{<:}} \Delta(\texttt{X}) \qquad \text{(S-Var)}$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{D<}\overline{\texttt{S}}\texttt{> \{...\}}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \mathrel{\texttt{<:}} [\overline{\texttt{T}}/\overline{\texttt{X}}]\texttt{D<}\overline{\texttt{S}}\texttt{>}} \qquad \text{(S-Class)}$$

$$\Delta \vdash * \mathrel{\texttt{<:}} \texttt{T} \qquad \text{(S-Bot)}$$

$$\frac{\texttt{C} \trianglelefteq \texttt{D}}{\Delta \vdash \texttt{C} \mathrel{\texttt{<:}} \texttt{D}} \qquad \text{(S-Raw)}$$

$$\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \mathrel{\texttt{<:}} \texttt{C} \qquad \text{(S-Cooked/Raw)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{D \{...\}} \\ CT(\texttt{D}) = \texttt{class D<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>} \triangleleft \texttt{N \{...\}} \\ \#(\overline{\texttt{Y}}) = \#(\overline{*}) \end{array}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \mathrel{\texttt{<:}} \texttt{D<}\overline{*}\texttt{>}} \qquad \text{(S-RawSuper)}$$

Unsafe Subtyping:

$$\frac{\Delta \vdash \texttt{S} \mathrel{\texttt{<:}} \texttt{T}}{\Delta \vdash \texttt{S} \mathrel{\texttt{<:}_?} \texttt{T}} \qquad \text{(SU-Safe)}$$

$$\frac{\Delta \vdash \texttt{C} \mathrel{\texttt{<:}} \texttt{D} \qquad \textit{unchecked warning}}{\Delta \vdash \texttt{C} \mathrel{\texttt{<:}_?} \texttt{D<}\overline{\texttt{T}}\texttt{>}} \qquad \text{(SU-Unsafe)}$$

Well-formed types:

$$\Delta \vdash \texttt{Object}\ \text{ok} \qquad \text{(WF-Object)}$$

$$\frac{\texttt{X} \in dom(\Delta)}{\Delta \vdash \texttt{X}\ \text{ok}} \qquad \text{(WF-Var)}$$

$$\frac{\begin{array}{c} CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}} \\ \Delta \vdash \overline{\texttt{T}}\ \text{ok} \qquad \Delta \vdash \overline{\texttt{T}} \mathrel{\texttt{<:}} [\overline{\texttt{T}}/\overline{\texttt{X}}]\overline{\texttt{N}} \end{array}}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>}\ \text{ok}} \qquad \text{(WF-Class)}$$

$$\Delta \vdash *\ \text{ok} \qquad \text{(WF-Bot)}$$

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{N \{...\}}}{\Delta \vdash \texttt{C}\ \text{ok}} \qquad \text{(WF-Raw)}$$

Figure 5: Raw FGJ: Subtyping and Type Well-fomedness Rules

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 $\texttt{C<}\overline{\texttt{T}}\texttt{>}$ is subtype of the corresponding raw type `C` (S-Cooked/Raw). Moreover, we need to add a rule for subtyping derived from subclassing:

$$\frac{CT(\texttt{C}) = \texttt{class C<}\overline{\texttt{X}} \triangleleft \overline{\texttt{N}}\texttt{>} \triangleleft \texttt{D\{...\}} \qquad CT(\texttt{D}) = \texttt{class D<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>} \triangleleft \texttt{N\{...\}} \qquad \#(\overline{\texttt{Y}}) = \#(\overline{*})}{\Delta \vdash \texttt{C<}\overline{\texttt{T}}\texttt{>} \mathrel{\texttt{<:}} \texttt{D<}\overline{*}\texttt{>}} \qquad \text{(S-RawSuper)}$$

As we already discussed, when the superclass is raw `D`, parameterized types from the subclass are subtype of $\texttt{D<}\overline{*}\texttt{>}$ rather than `D`.

The judgment of unsafe subtyping is written $\Delta \vdash \texttt{S} \mathrel{\texttt{<:}_?} \texttt{T}$; 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 $\texttt{D<}\overline{\texttt{T}}\texttt{>}$ (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 (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:

$$\frac{\begin{array}{c} \Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad mtype(\texttt{m}, bound_\Delta(\texttt{T}_0)) = \texttt{<}\overline{\texttt{Y}} \triangleleft \overline{\texttt{P}}\texttt{>}\overline{\texttt{U}} \rightarrow \texttt{U} \\ \Delta \vdash \overline{\texttt{V}}\ \text{ok} \qquad \Delta \vdash \overline{\texttt{V}} \mathrel{\texttt{<:}} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{P}} \qquad \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}} \mathrel{\texttt{<:}_?} [\overline{\texttt{V}}/\overline{\texttt{Y}}]\overline{\texttt{U}} \end{array}}{\Delta;\Gamma \vdash \texttt{e}_0\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in [\overline{\texttt{V}}/\overline{\texttt{Y}}]\texttt{U}} \qquad \text{(T-Invk)}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{T}_0 \qquad \mathit{mtyperaw}(\texttt{m}, |\texttt{T}_0|_\Delta) = \overline{\texttt{C}}\to\texttt{C} \qquad \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}} <:_? \overline{\texttt{C}}}{\Delta;\Gamma \vdash \texttt{e}_0\texttt{.m(}\overline{\texttt{e}}\texttt{)} \in \texttt{C}} \quad \text{(T-InvkR)}$$

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()` 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 `*`.

$$\frac{\Delta \vdash \texttt{C<}\overline{\texttt{U}}\texttt{>}\ \text{ok} \qquad \mathit{cargtype}(\texttt{C<}\overline{\texttt{U}}\texttt{>}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \qquad \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}} <:_? \overline{\texttt{T}}}{\Delta;\Gamma \vdash \texttt{new C<}\overline{\texttt{U}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in \texttt{C<}\overline{\texttt{U}}\texttt{>}} \quad \text{(T-New)}$$

$$\frac{\Delta \vdash \texttt{C<}\overline{\texttt{*}}\texttt{>}\ \text{ok} \qquad \mathit{cargtype}(\texttt{C}) = \overline{\texttt{T}}\ \overline{\texttt{f}} \qquad \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \Delta \vdash \overline{\texttt{S}} <:_? \overline{\texttt{T}}}{\Delta;\Gamma \vdash \texttt{new C(}\overline{\texttt{e}}\texttt{)} \in \texttt{C<}\overline{\texttt{*}}\texttt{>}} \quad \text{(T-NewR)}$$

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.

$$\frac{\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{*} \qquad \textit{stupid warning}}{\Delta;\Gamma \vdash \texttt{e}_0\texttt{.f}_i \in \texttt{*}} \quad \text{(T-Bot-Field)}$$

$$\frac{\Delta;\Gamma \vdash \texttt{e}_0 \in \texttt{*} \qquad \Delta \vdash \overline{\texttt{V}}\ \text{ok} \qquad \Delta;\Gamma \vdash \overline{\texttt{e}} \in \overline{\texttt{S}} \qquad \textit{stupid warning}}{\Delta;\Gamma \vdash \texttt{e}_0\texttt{.m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{e}}\texttt{)} \in \texttt{*}} \quad \text{(T-Bot-Invk)}$$

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 $\texttt{new N(}\overline{\texttt{e}}\texttt{)}$. 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 $F_{\leq}$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.)

$$\frac{\mathit{mbodyraw}(\texttt{m}, \texttt{N}) = (\overline{\texttt{x}}, \texttt{e})}{\texttt{new N(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)} \longrightarrow [\overline{\texttt{d}}/\overline{\texttt{x}}, \texttt{new N(}\overline{\texttt{e}}\texttt{)/this}]\texttt{e}} \quad \text{(R-InvkR)}$$

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 $\Delta;\Gamma \vdash \texttt{e} \in \texttt{T}$ without *unchecked warning* and $\texttt{e} \longrightarrow \texttt{e}'$, then $\Delta;\Gamma \vdash \texttt{e}' \in \texttt{T}'$ without *unchecked warning*, for some $\texttt{T}'$ such that $\Delta \vdash \texttt{T}' <: \texttt{T}$.

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

(1) If `e` includes $\texttt{new N}_0\texttt{(}\overline{\texttt{e}}\texttt{).f}$ as a subexpression, then $\mathit{fields}(\texttt{N}_0) = \overline{\texttt{T}}\ \overline{\texttt{f}}$ and $\texttt{f} \in \overline{\texttt{f}}$.

(2) If `e` includes $\texttt{new N}_0\texttt{(}\overline{\texttt{e}}\texttt{).m<}\overline{\texttt{V}}\texttt{>(}\overline{\texttt{d}}\texttt{)}$ as a subexpression, then $\mathit{mbody}(\texttt{m<}\overline{\texttt{V}}\texttt{>}, \texttt{N}_0) = (\overline{\texttt{x}}, \texttt{e}_0)$ and $\#(\overline{\texttt{x}}) = \#(\overline{\texttt{d}})$.

(3) If `e` includes $\texttt{new N}_0\texttt{(}\overline{\texttt{e}}\texttt{).m(}\overline{\texttt{d}}\texttt{)}$ as a subexpression, then $\mathit{mbodyraw}(\texttt{m}, \texttt{N}_0) = (\overline{\texttt{x}}, \texttt{e}_0)$ and $\#(\overline{\texttt{x}}) = \#(\overline{\texttt{d}})$.

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

Expression typing:

$$\Delta;\Gamma \vdash \mathtt{x} \in \Gamma(\mathtt{x}) \qquad \text{(T-VAR)}$$

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{fields}(\mathit{bound}_\Delta(\mathtt{T}_0)) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in \mathtt{T}_i} \qquad \text{(T-FIELD)}$$

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{mtype}(\mathtt{m}, \mathit{bound}_\Delta(\mathtt{T}_0)) = \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U} \\ \Delta \vdash \overline{\mathtt{V}}\ \text{ok} \qquad \Delta \vdash \overline{\mathtt{V}} \mathrel{<:} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{P}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:_?} [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\overline{\mathtt{U}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}\mathtt{<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}}) \in [\overline{\mathtt{V}}/\overline{\mathtt{Y}}]\mathtt{U}} \qquad \text{(T-INVK)}$$

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \\ \mathit{mtyperaw}(\mathtt{m}, |\mathtt{T}_0|_\Delta) = \overline{\mathtt{C}} \rightarrow \mathtt{C} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:_?} \overline{\mathtt{C}}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in \mathtt{C}} \qquad \text{(T-INVKR)}$$

$$\dfrac{\begin{array}{c}\Delta \vdash \mathtt{C<}\overline{\mathtt{U}}\mathtt{>}\ \text{ok} \qquad \mathit{cargtype}(\mathtt{C<}\overline{\mathtt{U}}\mathtt{>}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:_?} \overline{\mathtt{T}}\end{array}}{\Delta;\Gamma \vdash \mathtt{new\ C<}\overline{\mathtt{U}}\mathtt{>}(\overline{\mathtt{e}}) \in \mathtt{C<}\overline{\mathtt{U}}\mathtt{>}} \qquad \text{(T-NEW)}$$

$$\dfrac{\begin{array}{c}\Delta \vdash \mathtt{C<}\overline{\mathtt{*}}\mathtt{>}\ \text{ok} \qquad \mathit{cargtype}(\mathtt{C}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \Delta \vdash \overline{\mathtt{S}} \mathrel{<:_?} \overline{\mathtt{T}}\end{array}}{\Delta;\Gamma \vdash \mathtt{new\ C}(\overline{\mathtt{e}}) \in \mathtt{C<}\overline{\mathtt{*}}\mathtt{>}} \qquad \text{(T-NEWR)}$$

$$\dfrac{\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{T}_0 \mathrel{<:} \mathtt{N}}{\Delta;\Gamma \vdash (\mathtt{N})\mathtt{e}_0 \in \mathtt{N}} \qquad \text{(T-UCAST)}$$

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \Delta \vdash \mathtt{N}\ \text{ok} \\ \Delta \vdash \mathtt{N} \mathrel{<:} \mathit{bound}_\Delta(\mathtt{T}_0) \qquad \mathit{dcast}(\mathtt{N}, \mathit{bound}_\Delta(\mathtt{T}_0))\end{array}}{\Delta;\Gamma \vdash (\mathtt{N})\mathtt{e}_0 \in \mathtt{N}} \qquad \text{(T-DCAST)}$$

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in \mathtt{T}_0 \qquad \mathit{bound}_\Delta(\mathtt{T}_0) = \mathtt{D<}\overline{\mathtt{U}}\mathtt{>} \\ \mathtt{N} = \mathtt{C<}\overline{\mathtt{T}}\mathtt{>} \qquad \Delta \vdash \mathtt{N}\ \text{ok} \qquad \mathtt{C} \ntrianglelefteq \mathtt{D} \qquad \mathtt{D} \ntrianglelefteq \mathtt{C} \\ \textit{stupid warning}\end{array}}{\Delta;\Gamma \vdash (\mathtt{N})\mathtt{e}_0 \in \mathtt{N}} \qquad \text{(T-SCAST)}$$

$$\dfrac{\Delta;\Gamma \vdash \mathtt{e}_0 \in * \qquad \textit{stupid warning}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{f}_i \in *} \qquad \text{(T-BOT-FIELD)}$$

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in * \qquad \Delta \vdash \overline{\mathtt{V}}\ \text{ok} \\ \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \qquad \textit{stupid warning}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}(\overline{\mathtt{e}}) \in *} \qquad \text{(T-BOT-INVK)}$$

$$\dfrac{\begin{array}{c}\Delta;\Gamma \vdash \mathtt{e}_0 \in * \qquad \Delta;\Gamma \vdash \overline{\mathtt{e}} \in \overline{\mathtt{S}} \\ \textit{stupid warning}\end{array}}{\Delta;\Gamma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) \in *} \qquad \text{(T-BOT-INVKR)}$$

Method typing:

$$\dfrac{\begin{array}{c}\mathit{mtype}(\mathtt{m}, \mathtt{N}) = \mathtt{<}\overline{\mathtt{Z}} \triangleleft \overline{\mathtt{Q}}\mathtt{>}\overline{\mathtt{U}} \rightarrow \mathtt{U}_0 \text{ implies} \\ \overline{\mathtt{P}},\overline{\mathtt{T}} = [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}](\overline{\mathtt{Q}},\overline{\mathtt{U}}) \text{ and } \Delta \vdash \mathtt{T}_0 \mathrel{<:} [\overline{\mathtt{Y}}/\overline{\mathtt{Z}}]\mathtt{U}_0\end{array}}{\mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T}_0)}$$

$$\dfrac{\begin{array}{c}\mathit{mtyperaw}(\mathtt{m}, \mathtt{C}) = \overline{\mathtt{D}} \rightarrow \mathtt{D}_0 \text{ implies} \\ \overline{\mathtt{D}} = \overline{\mathtt{C}} \text{ and } \Delta \vdash \mathtt{C}_0 \mathrel{<:} \mathtt{D}_0\end{array}}{\mathit{override}(\mathtt{m}, \mathtt{C}, \overline{\mathtt{C}} \rightarrow \mathtt{C}_0)}$$

$$\dfrac{\begin{array}{c}\Delta = \overline{\mathtt{X}}\mathrel{<:}\overline{\mathtt{N}}, \overline{\mathtt{Y}}\mathrel{<:}\overline{\mathtt{P}} \qquad \Delta \vdash \overline{\mathtt{T}}, \mathtt{T}, \overline{\mathtt{P}}\ \text{ok} \\ \Delta; \overline{\mathtt{x}} : \overline{\mathtt{T}}, \mathtt{this} : \mathtt{C<}\overline{\mathtt{X}}\mathtt{>} \vdash \mathtt{e}_0 \in \mathtt{S} \qquad \Delta \vdash \mathtt{S} \mathrel{<:} \mathtt{T} \\ \mathit{CT}(\mathtt{C}) = \mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\ldots\} \\ \mathit{override}(\mathtt{m}, \mathtt{N}, \mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>}\overline{\mathtt{T}} \rightarrow \mathtt{T})\end{array}}{\mathtt{<}\overline{\mathtt{Y}} \triangleleft \overline{\mathtt{P}}\mathtt{>\ T\ m(}\overline{\mathtt{T}}\ \overline{\mathtt{x}}\mathtt{)\{\ return\ e}_0\mathtt{;\ \}\ OK\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>}} \qquad \text{(T-METHOD)}$$

Class typing:

$$\dfrac{\begin{array}{c}\overline{\mathtt{X}}\mathrel{<:}\overline{\mathtt{N}} \vdash \overline{\mathtt{N}}, \overline{\mathtt{T}}, \mathtt{N}\ \text{ok} \\ \mathit{cargtype}(\mathtt{N}) = \overline{\mathtt{U}}\ \overline{\mathtt{g}} \qquad \overline{\mathtt{M}}\ \mathtt{OK\ IN\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \\ \mathtt{K = C(}\overline{\mathtt{U}}\ \overline{\mathtt{g}}\mathtt{,\ }\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{)\{\ super(}\overline{\mathtt{g}}\mathtt{);\ this.}\overline{\mathtt{f}}\mathtt{=}\overline{\mathtt{f}}\mathtt{;\ \}}\end{array}}{\mathtt{class\ C<}\overline{\mathtt{X}} \triangleleft \overline{\mathtt{N}}\mathtt{>} \triangleleft \mathtt{N}\ \{\overline{\mathtt{T}}\ \overline{\mathtt{f}}\mathtt{;\ K}\ \overline{\mathtt{M}}\}\ \mathtt{OK}} \qquad \text{(T-CLASS)}$$

Figure 6: Raw FGJ: Typing Rules

$$\dfrac{\mathit{fields}(\mathtt{N}) = \overline{\mathtt{T}}\ \overline{\mathtt{f}}}{\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).f}_i \longrightarrow \mathtt{e}_i} \qquad \text{(R-FIELD)}$$

$$\dfrac{\mathit{mbody}(\mathtt{m<}\overline{\mathtt{V}}\mathtt{>}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e}_0)}{\begin{array}{l}\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).m<}\overline{\mathtt{V}}\mathtt{>(}\overline{\mathtt{d}}\mathtt{)} \\ \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}_0\end{array}} \qquad \text{(R-INVK)}$$

$$\dfrac{\mathit{mbodyraw}(\mathtt{m}, \mathtt{N}) = (\overline{\mathtt{x}}, \mathtt{e})}{\begin{array}{l}\mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{).m(}\overline{\mathtt{d}}\mathtt{)} \\ \longrightarrow [\overline{\mathtt{d}}/\overline{\mathtt{x}}, \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)/this}]\mathtt{e}\end{array}} \qquad \text{(R-INVKR)}$$

$$\dfrac{\emptyset \vdash \mathtt{N} \mathrel{<:} \mathtt{P}}{\mathtt{(P)(new\ N(}\overline{\mathtt{e}}\mathtt{))} \longrightarrow \mathtt{new\ N(}\overline{\mathtt{e}}\mathtt{)}} \qquad \text{(R-CAST)}$$

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 $\exists$`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

[1] Ole Agesen, Stephen N. Freund, and John C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 49–65, Atlanta, GA, October 1997.

[2] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. GJ specification. Manuscript, May 1998. Available through `http://cm.bell-labs.com/cm/cs/who/wadler/gj/`.

[3] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM SIGPLAN Notices, volume 33, number 10, pages 183–200, Vancouver, BC, October 1998.

[4] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John Mitchell. F-bounded quantification for object-oriented programming. In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, September 1989.

[5] Luca Cardelli, Simone Martini, John C. Mitchell, and Andre Scedrov. An extension of system F with subtyping. *Information and Computation*, 109(1–2):4–56, 1994. Preliminary version in TACS '91 (Sendai, Japan, pp. 750–770).

[6] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, December 1985.

[7] Robert Cartwright and Guy L. Steele Jr. Compatible genericity with run-time types for the Java programming language. In Craig Chambers, editor, *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, SIGPLAN Notices volume 33 number 10, pages 201–215, Vancouver, BC, October 1998. ACM.

[8] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972. A summary appeared in the Proceedings of the Second Scandinavian Logic Symposium (J.E. Fenstad, editor), North-Holland, 1971 (pp. 63–92).

[9] Atsushi Igarashi and Benjamin C. Pierce. On inner classes. In Elisa Bertino, editor, *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 129–153, Cannes, France, June 2000. Springer-Verlag. Extended version will appear in *Information and Computation*.

[10] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Linda M. Northrop, editor, *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOSPLA'99)*, ACM SIGPLAN Notices, volume 34, number 10, pages 132–146, Denver, CO, October 1999. ACM Press.

[11] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. A recipe for raw types. Technical report, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 2001. In preparation.

[12] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential types. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proceedings of 12th ACM Symposium on Principles of Programming Languages,* 1985.

[13] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 132–145, Paris, France, January 1997.

[14] Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, pages 146–159, Paris, France, January 1997.

[15] Benjamin C. Pierce. Bounded quantification with bottom. Technical Report 492, Computer Science Department, Indiana University, 1997.

[16] John Reynolds. Towards a theory of type structure. In *Proc. Colloque sur la Programmation*, pages 408–425, New York, 1974. Springer-Verlag LNCS 19.

[17] Mirko Viroli and Antonio Natali. Parametric polymorphism in Java: an approach to translation based on reflective features. In Doug Lea, editor, *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, ACM SIGPLAN Notices, volume 35, number 10, pages 146–165, Minneapolis, MN, October 2000. ACM Press.

[18] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, November 1994.