JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2007

Vol. 7, No. 5, Special Issue: Workshop on FTfJP 07, June 2008

Adding Type Constructor Parameterization to Java

Vincent Cremet, France
Philippe Altherr, Switzerland

We present a generalization of Java's parametric polymorphism that enables parameterization of classes and methods by type constructors, i.e., functions from types to types. Our extension is formalized as a calculus called FGJ_ω. It is implemented in a prototype compiler and its type system is proven safe and decidable. We describe our extension and motivate its introduction in an object-oriented context through two examples: the definition of generic data-types with binary methods and the definition of generalized algebraic data-types. The Coq proof assistant was used to formalize FGJ_ω and to mechanically check its proof of type safety.

INTRODUCTION

Most mainstream statically typed programming languages (Java, C++, Ada) let the programmer parameterize data structures and algorithms by *types*. The general term for this mechanism is parametric polymorphism, which is called "generics" in Java and "templates" in C++. Parametric polymorphism allows the same piece of code to be used with different type instantiations. Some languages, like Haskell [Jon03], additionally let the programmer parameterize code by *type constructors*, i.e., functions from types to types. One typical application of this feature is to parameterize a piece of code by a generic data-type, i.e., a data-type which is itself parameterized by a type. Although this mechanism has been widely recognized as useful in Haskell, for example to represent monads [Wad92] as a library, there is little work about the introduction of type constructor parameterization in Java-like languages. In this paper we present a type-safe design for Java. Our syntax partially inspired the independent integration of type constructor parameterization in the Scala [Ot07] compiler where the feature is called type constructor polymorphism [MPO07].

A *type constructor*, or type operator, is a function that takes a list of types and returns a type. For instance, the Java class `List` of the standard library defines a type constructor; `List` can be applied to a type T with the syntax `List<`T`>` to denote the type of lists of Ts. In Java, classes and methods can be parameterized by types but not by type constructors. Our generalization of Java's parametric polymorphism lifts this restriction.

Our design is introduced and explained in the next two sections through two different examples: the definition of generic data-types with binary methods and the definition of generalized algebraic data-types. Section 1 shows how type constructor

Cite this article as follows: Vincent Cremet, Philippe Altherr, "Adding Type Constructor Parameterization to Java", in *Journal of Object Technology*, vol. 7, no. 5, Special Issue: Workshop on FTfJP 07, June 2008, pp. 25-65
http://www.jot.fm/issues/issue_2008_06/article2/

parameterization lets us improve the classical solution to represent binary methods in Java; the improved solution applies to the case where the binary method belongs to a generic class. Section 2 shows how to enhance the Visitor design-pattern with type constructor parameterization in order to implement generalized algebraic data-types. Section 3 describes our FGJ_ω calculus (pronounce "FGJ-omega"), which formalizes our design for type constructor parameterization in Java. We have proven that FGJ_ω's type system is both safe and decidable. Section 4 explains the interesting and novel aspects of the proofs. The proofs are detailed in appendix and the proof of type safety has been additionally formalized in a proof assistant. Section 5 reviews related work and Section 6 concludes with a summary of our contributions and mentions possible continuations.

A prototype FGJ_ω compiler, which consists of a type-checker and an interpreter, along with the formalization and the proofs developed in the Coq proof assistant [Pro04] are available on the FGJ_ω home page [ACa].

1 GENERIC DATA-TYPES WITH BINARY METHODS

Binary methods [BCC+96] are a well-known challenge for object-oriented programming. This section first presents the problem posed by binary methods. Then, it describes a classical technique based on parametric polymorphism that can sometimes be used to solve the problem. Finally, it shows how to generalize this technique to generic data-types with the help of type constructor parameterization.

Binary methods

A binary method is a method whose signature contains occurrences of the current class in contravariant positions, for example as the type of a parameter. The classical examples of problematic binary methods are comparison methods where the receiver and the arguments should be of the same type. When such methods have to be implemented in a subclass, it would be necessary to refine the type of their parameter, which is not allowed (actually it would be unsafe).

In the code below, we illustrate the problem with the binary methods `lessThan` and `greaterThan` of the class `Ord`. To implement `lessThan` in `OrdInt`, it would be necessary to change the type of the parameter `that` from `Ord` to `OrdInt` to have access to the field `i` of `that` but this is not a legal overriding. If it was, it would be unsafe, as shown by the last line of our example.

```
abstract class Ord {
  abstract boolean lessThan(Ord that);
  boolean greaterThan(Ord that) { return that.lessThan(this); }
}
class OrdInt extends Ord { int i;
```

```
  @Override // compile-time error (method overrides nothing)
  boolean lessThan(OrdInt that) { return this.i < that.i; }
}
Ord o1 = new OrdInt();
Ord o2 = new AnotherOrd();
o1.lessThan(o2);  // runtime error (if code was allowed to compile)
```

Classical solution

The classical technique is to parameterize the base class with a type `Self`, also called its *self type*, which represents the exact type of the current object (concrete subclasses are expected to instantiate `Self` with their own type). The type `Self` is bounded by the type of the base class and occurrences of the base class in the signature of binary methods are replaced with `Self`. If, like in our example, this leads to code where `this` occurs in places where an instance of `Self` is expected, a method `self` of type `Self` must be added to the base class and the offending occurrences of `this` must be replaced with calls to `self`. Concrete subclasses are expected to implement `self` by returning `this`.

```
abstract class Ord<Self extends Ord<Self>> {
  abstract Self self();
  abstract boolean lessThan(Self that);
  boolean greaterThan(Self that) { return that.lessThan(self()); }
}
class OrdInt extends Ord<OrdInt> { int i;
  @Override OrdInt self() { return this; }
  @Override boolean lessThan(OrdInt that) { return this.i < that.i; }
}
Ord<OrdInt>     o1 = new OrdInt();
Ord<AnotherOrd> o2 = new AnotherOrd();
o1.lessThan(o2);  // compile-time error (as expected)
```

Generalizing the problem and the solution

The above technique does not always directly apply when the base class is generic. To see why, we consider a class `ICollection`, representing immutable collections of objects, that is parameterized by the type `X` of its elements. For the purpose of our demonstration a collection declares just two methods: `append`, which merges two collections into a new one, and `flatMap`, which applies a function from elements to collections to all the elements of a collection and merges the returned collections into a new one.

```
// functions from X to Y
```

```
abstract class Function<X,Y> { abstract Y apply(X x); }

abstract class ICollection<X> {
  abstract     ICollection<X> append(ICollection<X> that);
  abstract <Y> ICollection<Y> flatMap(Function<X,ICollection<Y>> f);
}
```

Both methods are binary methods because the current class occurs in the type of their parameter. A simple and efficient implementation of `append` and `flatMap` for immutable linked lists is illustrated below. Unfortunately this implementation is possible only if the occurrences of `ICollection` in their signatures can be replaced with `IList`, which is allowed for the return types but not for the parameter types.

```
class IList<X> extends ICollection<X> {
           IList()        { ... } // constructs an empty list
  boolean  isEmpty()      { ... } // tests emptiness
  X        head()         { ... } // gets first element
  IList<X> tail()         { ... } // gets all elements but the first
  IList<X> add(X that)    { ... } // adds an element at the head
  @Override // compile-time error (method overrides nothing)
  IList<X> append(IList<X> that) {
    return isEmpty() ? that : tail().append(that).add(head());
  }
  @Override // compile-time error (method overrides nothing)
  <Y> IList<Y> flatMap(Function<X,IList<Y>> f) {
    return isEmpty() ? new IList<Y>()
                     : f.apply(head()).append(tail().<Y>flatMap(f));
  }
}
```

Applying the same technique as for the class `Ord` by replacing every occurrence of `ICollection<X>` with a type parameter `Self` solves the problem for `append` but not for `flatMap`. It is unclear with what `ICollection<Y>` should be replaced. Replacing it by `Self` would not work as it would make the method less general.

```
abstract class ICollection<Self extends ICollection<Self,X>,X> {
  abstract     Self append(Self that);
  abstract <Y> ?    flatMap(Function<X,?> f);
}
class IList<X> extends ICollection<IList<X>,X> {/* same as above */}
```

For `flatMap`, we need more than a self *type*, we need a self *type constructor*, i.e., `Self` should represent a type constructor instead of a type. This is expressed in the following definition, where `Self<Z> extends ICollection<Self,Z>` declares a

type constructor expecting one parameter `Z` such that when `Self` is applied to some type `Z` it returns a subtype of `ICollection<Self,Z>`.

```
abstract class ICollection<Self<Z> extends ICollection<Self,Z>,X> {
  abstract     Self<X> append(Self<X> that);
  abstract <Y> Self<Y> flatMap(Function<X,Self<Y>> f);
}
```

This lets us define the class `IList` with `append` and `flatMap` methods that indeed override the methods of `ICollection`.

```
class IList<X> extends ICollection<IList,X> { ...
  @Override     IList<X> append(IList<X> that)           { ... }
  @Override <Y> IList<Y> flatMap(Function<X,IList<Y>> f) { ... }
}
```

2 GENERALIZED ALGEBRAIC DATA-TYPES

An *algebraic* data-type is a type that is defined by the union of different cases. Operations on such types are usually implemented through pattern-matching. Classically, in Java, algebraic data-types are implemented by an abstract base class and a subclass for each case and operations on such types by the Visitor design pattern [GHJV94].

When an algebraic data-type is generic and different cases instantiate its type parameter with different types, it is called a *generalized algebraic data-type* (GADT) [KR05]. In this section we show that the Visitor design pattern is not expressive enough to implement operations on a GADT without resorting to type downcasts or to code duplication. Our contribution is to show that visitors can faithfully implement operations on GADTs if they are augmented with a type constructor parameter.

Limitations of visitors

The following example implements an algebraic data-type `Expr` representing expressions of a simple programming language.

```
abstract class Expr {}
class IntLit  extends Expr { int x; }                 // 0 | 1 | ...
class BoolLit extends Expr { boolean x; }             // false | true
class Plus    extends Expr { Expr x; Expr y; }        // x + y
class Compare extends Expr { Expr x; Expr y; }        // x < y
class If      extends Expr { Expr x; Expr y; Expr z; }// x ? y : z
```

Although we have omitted them for space reason, we assume that each subclass has a constructor that initializes all its fields. Thus, we can create values representing valid expressions but unfortunately also values representing invalid ones.

```
new Plus(new IntLit(2), new IntLit(3))          // 2 + 3
new Plus(new IntLit(2), new BoolLit(true))      // 2 + true
```

Forbidding the creation of invalid expressions is easy. The classical solution is to augment the class `Expr` with a type parameter `X` representing the kind of values that the expression evaluates to.

```
abstract class Expr<X> {}
class IntLit  extends Expr<Integer> { int x; }
class BoolLit extends Expr<Boolean> { boolean x; }
class Plus    extends Expr<Integer> { Expr<Integer> x, y; }
class Compare extends Expr<Boolean> { Expr<Integer> x, y; }
class If<Y>   extends Expr<Y>       { Expr<Boolean> x; Expr<Y> y,z;}
```

Since different cases instantiate its type parameter `X` with different types, the type `Expr` is by definition a *generalized algebraic data type*. Values representing valid expressions may still be built but values representing invalid ones are now rejected.

```
new Plus(new IntLit(2), new IntLit(3))      // 2 + 3
new Plus(new IntLit(2), new BoolLit(true))  // compile-time error
```

Operations on algebraic data-types like `Expr` can be implemented with the Visitor design pattern [GHJV94]. This pattern requires the definition of a class `Visitor` that declares an abstract method for each case of the data-type and that is parameterized by the return type `R` of the operation.

```
abstract class Visitor<R> {
  abstract     R caseIntLit  (IntLit  expr);
  abstract     R caseBoolLit (BoolLit expr);
  abstract     R casePlus    (Plus    expr);
  abstract     R caseCompare (Compare expr);
  abstract <Y> R caseIf      (If<Y>   expr);
}
```

The pattern also requires that the base class `Expr` declares a new method `accept` that applies a visitor to the expression. The method must be implemented in each subclass. For conciseness, we give only two representative subclasses.

```
abstract class Expr<X> {
  abstract <R> R accept(Visitor<R> v);
}
class IntLit extends Expr<Integer> { int x;
  <R> R accept(Visitor<R> v) { return v.caseIntLit(this); }
}
class Plus extends Expr<Integer> { Expr<Integer> x, y;
  <R> R accept(Visitor<R> v) { return v.casePlus(this); }
}
```

We can now implement a first operation that converts an expression into a string.

```
class PrintVisitor extends Visitor<String> {
  <X> String print(Expr<X> e) { return e.accept(this); }
  String caseIntLit(IntLit expr) { return String.valueOf(expr.x); }
  String casePlus(Plus expr) {
    return print(expr.x) + "+" + print(expr.y);
  }
  <Y> String caseIf(If<Y> expr) {
    return print(expr.x) + "?" + print(expr.y) + ":" + print(expr.z);
  }
  ...
}
```

We can also implement an operation that evaluates an expression but not without resorting to some downcasts because the result type of that operation depends on the actual type of the evaluated expression.

```
class EvalVisitor extends Visitor<Object> {
  <X> Object eval(Expr<X> e) { return e.accept(this); }
  Object caseIntLit(IntLit expr) { return expr.x; }
  Object casePlus(Plus expr) {
    return (Integer)eval(expr.x) + (Integer)eval(expr.y);
  }
  <Y> Object caseIf(If<Y> expr) {
    return (Boolean)eval(expr.x) ? eval(expr.y) : eval(expr.z);
  }
  ...
}
```

Admittedly, there is a way of implementing an evaluator based on visitors without resorting to downcasts. The solution is to define a new visitor `Visitor2` where each case returns a value of the appropriate type.

```
abstract class Visitor2 {
  abstract     Integer caseIntLit  (IntLit  expr);
  abstract     Boolean caseBoolLit (BoolLit expr);
  abstract     Integer casePlus    (Plus    expr);
  abstract     Boolean caseCompare (Compare expr);
  abstract <Y> Y       caseIf      (If<Y>   expr);
}
```

This solution is not very satisfactory; it involves a lot of code duplication. The class `Visitor2` is almost identical to the class `Visitor`, the difference lying only in the return type of its methods. The class `Expr` and its subclasses must also be augmented with a new method `accept2` whose declaration and implementations are again almost identical to the ones of the method `accept`. This is not scalable as each time an operation that returns a new kind of type needs to be implemented, a lot of code has to be duplicated. Furthermore, this is only possible if the operation implementer may modify the classes implementing the data-type. So, a question arises: would it be possible to write a single "universal" visitor?

A universal visitor

A single visitor taking as a parameter a function `R` from types to types can replace both of our previous visitors. The return type of each case of this visitor is obtained by applying `R` to the evaluation type of the case. The syntax `R<_>` indicates that `R` stands for a unary type constructor and that we are not interested in naming its parameter.

```
abstract class Visitor<R<_>> {
  abstract     R<Integer> caseIntLit  (IntLit  expr);
  abstract     R<Boolean> caseBoolLit (BoolLit expr);
  abstract     R<Integer> casePlus    (Plus    expr);
  abstract     R<Boolean> caseCompare (Compare expr);
  abstract <Y> R<Y>       caseIf      (If<Y>   expr);
}
```

The method `accept` corresponding to this visitor is similarly parameterized by a function `R` from types to types.

```
abstract class Expr<X> {
  abstract <R<_>> R<X> accept(Visitor<R> v);
}
class IntLit extends Expr<Integer> { int x;
  <R<_>> R<Integer> accept(Visitor<R> v) { return v.caseIntLit(this); }
}
```

Anonymous Type Constructors

The print and the evaluation visitors can both be implemented by instantiating the visitor with the appropriate type function.

```
class PrintVisitor extends Visitor<<Y> => String> { ... }
class EvalVisitor  extends Visitor<<Y> => Y>      { ... }
```

For instance, the class `EvalVisitor` instantiates `R` with the type constructor `<Y> => Y`. Such an expression, called an *anonymous type constructor*, represents a function from types to types. The expression `<Y> => Y` represents the identity function, it takes a type parameter `Y` and returns the same type `Y`. Although in our examples anonymous type constructors are always used in class bounds, they can be used wherever type constructors are expected or in other words wherever type arguments are expected in standard Java.

In the class `EvalVisitor` the method `eval` can now be declared with a more precise return type.

```
<X> X eval(Expr<X> e) { return e.accept(this); }
```

One can check that the returned value corresponds to the declared return type. Since `e` is of type `Expr<X>` the type of `e.accept(this)` is `(<Y> => Y)<X>` (obtained from `R<X>` with `R` instantiated to `<Y> => Y`), which reduces in one step to `X`. Similarly one can check that the other methods of the visitor return values of the right type.

The following example demonstrates that visitors with more complex return types can also be implemented. It assumes that expressions may evaluate to multiple values and implements an evaluator that collects all the possible values of an expression.

```
class MultiEvalVisitor extends Visitor<<Y> => Set<Y>> {
  <X> Set<X> eval(Expr<X> e) { return e.accept(this); }
  Set<Integer> casePlus(Plus expr) {
    Set<Integer> set = new HashSet<Integer>();
    for (int a: eval(expr.x))
      for (int b: eval(expr.y)) set.add(a + b);
    return set;
  }
  ...
}
```

One could contend that our visitor is not truly universal. For example, it cannot implement (without downcasts) an evaluator that represents integers with `BigInteger`s and booleans with `Byte`s. However, this has more to do with the way

our data-type is defined than anything with our visitor. Indeed, the definition of our data-types ties integer and boolean expressions to the types `Integer` and `Boolean`. This has not to be. The data-type can be defined in such a way that integer and boolean expressions are not tied to any concrete type. With such a data-type and an accordingly adapted visitor, it is possible to reuse the same visitor to implement both an evaluator that represents integer and boolean expressions with `Integer`s and `Boolean`s and one that represents them with `BigInteger`s and `Byte`s. The implementation of such a data-type and visitor is given on FGJ$_\omega$'s home page [ACa].

3 FORMALIZATION: THE FGJ$_\omega$ CALCULUS

The FGJ$_\omega$ calculus is a formalization of our design for type constructor parameterization in Java. It is an extension of Featherweight Generic Java (FGJ) [IPW99], a core language for Java with a focus on generics. Our calculus enhances FGJ by replacing all the parameters representing types with parameters representing type constructors. Its syntax is given below. All the elements newly introduced or modified with respect to FGJ are highlighted. Like in FGJ, a program consists of a list of class declarations and a main term.

class declaration	D	::=	`class` C`<`$\overline{P}$`>` $\triangleleft$ C'`<`$\overline{K}$`>`$^?$ `{` $\overline{F}$ $\overline{M}$ `}`
field declaration	F	::=	T f`;`
method declaration	M	::=	`<`$\overline{P}$`>` T m`(`$\overline{T}$ $\overline{x}$`)` `{` `return` t`;` `}`
term	t	::=	x | t`.`f | t`.<`$\overline{K}$`>`m`(`$\overline{t}$`)` | `new` C`<`$\overline{K}$`>(`$\overline{t}$`)`
type parameter declaration	P	::=	X`<`$\overline{P}$`>` $\triangleleft$ C`<`$\overline{K}$`>`$^?$
type constructor	K	::=	X | C | `<`$\overline{P}$`>` `=>` T
type	T	::=	K`<`$\overline{K}$`>`

The metavariables C, m, f resp. range over class, method and field names; x and X resp. range over variables and type constructor variables. The notation $\overline{x}$ stands for the possibly empty sequence $x_1 \ldots x_n$ and $|\overline{x}|$ for its length. In the class and type parameter declarations, the symbol $\triangleleft$ replaces the keyword `extends` and the question mark indicates that the superclass and the upper bound are optional.

A type parameter declaration X`<`$\overline{P}$`>` $\triangleleft$ C`<`$\overline{K}$`>`$^?$ specifies that the parameter X represents a type constructor that accepts arguments conforming to the parameters $\overline{P}$ and that returns a type conforming to the bound C`<`$\overline{K}$`>` (if present) when it is applied to such arguments. A type constructor K is either a type variable X, a class name C, or an anonymous type constructor `<`$\overline{P}$`>` `=>` T that expects arguments conforming to the parameters $\overline{P}$ and that returns the type T. A type T consists of a type constructor K applied to a list of type constructors $\overline{K}$.

Except for the subtyping rules and the conformance rules of type constructors to type parameters, given below, FGJ$_\omega$'s type system and semantics are similar to FGJ's ones. The main difference is that subtyping includes the reduction of types so that a type like `(<X> => List<X>)<Integer>` is a subtype of `List<Integer>`. A complete formalization of FGJ$_\omega$'s type system can be found in the appendix.

$$\text{(S-Refl)}\quad \frac{}{\Delta \vdash T <: T}$$

$$\text{(S-Red}_\text{L}\text{)}\quad \frac{T \to T' \qquad \Delta \vdash T' <: U}{\Delta \vdash T <: U}$$

$$\text{(S-Red}_\text{R}\text{)}\quad \frac{U \to U' \qquad \Delta \vdash T <: U'}{\Delta \vdash T <: U}$$

$$\text{(S-Class)}\quad \frac{\texttt{class}\ C\texttt{<}\overline{P}\texttt{>} \lhd C'\texttt{<}\overline{K}'\texttt{>}\ \{\,\overline{F}\ \overline{M}\,\} \qquad \Delta \vdash C'\texttt{<}\overline{K}'\texttt{>}[\mathrm{vars}(\overline{P})\backslash\overline{K}] <: U}{\Delta \vdash C\texttt{<}\overline{K}\texttt{>} <: U}$$

$$\text{(S-Var)}\quad \frac{X\texttt{<}\overline{P}\texttt{>} \lhd C\texttt{<}\overline{K}'\texttt{>} \in \Delta \qquad \Delta \vdash C\texttt{<}\overline{K}'\texttt{>}[\mathrm{vars}(\overline{P})\backslash\overline{K}] <: U}{\Delta \vdash X\texttt{<}\overline{K}\texttt{>} <: U}$$

$$\text{(Poly-Sat)}\quad \frac{\forall i,\ \Delta \vdash K_i \in P_i[\mathrm{vars}(\overline{P})\backslash\overline{K}]}{\Delta \vdash (\overline{K}) \in (\overline{P})}$$

$$\text{(Sat)}\quad \frac{\overline{P}' = \mathrm{params}_\Delta(K) \qquad \Delta, \overline{P} \vdash \mathrm{vars}(\overline{P}) \in \overline{P}' \qquad \overline{P}_0 = \mathrm{erase}(\overline{P}) \qquad (\Delta, \overline{P}_0 \vdash K\texttt{<}\mathrm{vars}(\overline{P})\texttt{>} <: C\texttt{<}\overline{K}\texttt{>})^?}{\Delta \vdash K \in X\texttt{<}\overline{P}\texttt{>} \lhd C\texttt{<}\overline{K}\texttt{>}^?}$$

$$\text{where}\ \begin{cases} \mathrm{erase}(X\texttt{<}\overline{P}\texttt{>} \lhd C\texttt{<}\overline{K}\texttt{>}^?) &= X\texttt{<}\mathrm{erase}(\overline{P})\texttt{>} \lhd \texttt{none} \\ \mathrm{vars}(X\texttt{<}\overline{P}\texttt{>} \lhd C\texttt{<}\overline{K}\texttt{>}^?) &= X \end{cases}$$

The syntax allows types like `(<X> => X)<Integer>`, which is both a subtype and a supertype of `Integer` (thanks to S-Red$_\text{L}$, S-Red$_\text{R}$ and S-Refl). Such types are only marginally useful; for example to avoid repeating long and/or complex types like in `(<X> => Pair<X,X>)<MyLongAndComplexType>`. If judged undesirable, they could easily be forbidden through some simple syntax changes.

Our examples use some syntactic sugar; empty lists of type arguments and empty lists of type parameters are omitted: `Integer<>`, `X<> extends C<...>` and `<> => List<Integer>` are resp. abbreviated as `Integer`, `X extends C<...>` and `List<Integer>`. This implies that expressions like `Integer` and `List<Integer>` can denote both a type and a type constructor but there is never an ambiguity because it is always clear which one is expected from the context.

Thanks to this syntactic sugar any Java type also denotes a FGJ$_\omega$ type. Interestingly, some Java types like `List<Integer>` denote FGJ$_\omega$ types even without syntactic sugar but they are interpreted in a slightly different way. Indeed, in FGJ$_\omega$, `Integer` denotes a type constructor and not a type like in Java but this is fine because in FGJ$_\omega$ `List` expects a type constructor and not a type. Most Java types denote FGJ$_\omega$ types only thanks to the syntactic sugar : for example `Integer` and `List<List<Integer>>` resp. denote `Integer<>` and `List<<> => List<Integer>>`.

4 THEORETICAL STUDY OF FGJ$_\omega$

The type system of FGJ$_\omega$ is both safe and decidable. The detailed proofs of these properties can be found in appendix. Here we outline the main arguments of our reasoning. The Coq proof assistant [Pro04] was used to formalize FGJ$_\omega$ and to mechanically check most parts of its proof of type safety, including all the difficult ones. The formalization and the proofs are available on the FGJ$_\omega$ home page [ACa].

FGJ$_\omega$ is type safe

When we claim that FGJ$_\omega$ is type safe, we mean that well-typed FGJ$_\omega$ programs never go wrong or equivalently that well-typed programs either reduce forever or reduce to a value. More precisely, we prove that if p is a well-formed FGJ$_\omega$ program, t is its main expression and t reduces in zero or more steps to an irreducible term u, then u is a value.

The combination of subtyping and higher-order polymorphism makes type safety proofs difficult (as a demonstration of this claim see the complexity of a type safety proof [CG03] for a calculus similar to FGJ$_\omega$). The difficulty comes from the numerous dependencies that exist between subtyping, type reduction and type well-formedness. We present a proof technique that makes the proof tractable; the technique is simple but apparently new.

Our proof of type safety is based on a small-step operational semantics similar to the one of FGJ. Usually this property is proven by showing two theorems: a progress theorem (well-typed terms that are not values are reducible) and a subject-reduction theorem (term reduction preserves typing) [WF94]. The originality of our work is to prove these properties for a more general type system, called FGJ$_\Omega$ (pronounce "FGJ-big-omega"), with the same syntax and semantics as FGJ$_\omega$ but less constraining typing rules. The type safety of FGJ$_\omega$ is then deduced from the proof of type safety of FGJ$_\Omega$. This implies that we prove neither the progress nor the subject-reduction for FGJ$_\omega$ but we believe that this is not a big weakness. Indeed, the main interest of these properties is to prove type safety, which we do with our alternative technique.

Compared to the original type system, FGJ$_\Omega$ alleviates some constraints that were introduced to ensure decidability (for instance it permits the use of transitivity in subtyping). More importantly, while checking that a type application $K\texttt{<}\overline{K}\texttt{>}$ is well-formed, FGJ$_\Omega$ never enforces that the arguments $\overline{K}$ conform to the bounds of the parameters of K, only the conformance of arities is enforced. As we will explain later, it turns out that bound conformance has no impact on type safety. With the removal of this test type well-formedness no longer depends on subtyping and the proof becomes easier.

FGJ$_\omega$'s type system is decidable

We say that FGJ$_\omega$'s type system is *decidable* if there exists an algorithm that, for every input program, returns "yes" if the program is well-formed and "no" otherwise.

The most challenging issue with regard to decidability is subtyping. In FGJ, the decidability of subtyping essentially relies on the fact that there is no cycle in the class inheritance relation. This ensures that by following the superclass of a class, in the process of establishing that a type is a subtype of another, we always stop, either because we found an appropriate supertype or because we reached a class that has no superclass.

The proof that FGJ$_\omega$'s type system is decidable is a bit more complex because types sometimes need to be reduced in order to establish a subtyping judgment. So, there is a potential risk for the type-checker to be caught in an infinite reduction sequence. We show that types that are well-formed with respect to the conformance of arities (we call such types *well-kinded* types) can never be infinitely reduced (they are *strongly normalizable*). The proof of this property is similar to the proof of strong normalization for terms of the simply typed lambda calculus [Tai67, BBLS05]. This similarity is not surprising since well-kinded FGJ$_\omega$ types are almost isomorphic to well-typed lambda-terms. The only difference is the presence of bounds attached to FGJ$_\omega$'s type parameters and the fact that these bounds are reducible. For example, the anonymous type constructor `<X` $\triangleleft$ T`> =>` U reduces to `<X` $\triangleleft$ T'`> =>` U if T reduces to T'. There is no similar reduction step in the lambda-calculus. However, it is possible to translate well-kinded FGJ$_\omega$ types using a translation function $[\![\cdot]\!]$ into well-typed terms of the simply typed lambda-calculus with pairs, for which strong normalization still holds. We then prove that each time a FGJ$_\omega$ type T reduces to U in one step, the lambda-term $[\![T]\!]$ reduces to $[\![U]\!]$ in at least one step. Suppose now, by contradiction, there exists an infinite sequence of reductions starting from a well-kinded type T. Then there exists also an infinite sequence of reductions starting from the well-typed term $[\![T]\!]$, which contradicts the strong normalization of lambda-terms.

The main idea of the translation is to translate an anonymous type constructor into a pair whose first component corresponds to a translation of the same constructor but where its parameters have no bounds and whose second component is a translation of the sole parameters (with their bounds). The interesting cases of the translation are shown below. The function type($\overline{P}$) returns a suitable type for $\overline{P}$, and **fst** returns the first component of a pair.

$$\begin{array}{lcl} [\![\texttt{<}\overline{P}\texttt{>} \texttt{ => } T]\!] & = & (\lambda \text{vars}(\overline{P}) : \text{type}(\overline{P}).[\![T]\!], [\![\overline{P}]\!]) \\ [\![K\texttt{<}\overline{K}\texttt{>}]\!] & = & (\textbf{fst}\ [\![K]\!])\ [\![\overline{K}]\!] \end{array}$$

Bound conformance

The main difference between FGJ$_\omega$ and FGJ$_\Omega$ is that in the latter types where arguments do not conform to the bounds of the corresponding parameters are well-

formed. Intuitively, it seems that to ensure type safety such types should be rejected. Let us illustrate this with the following example.

```
class A<X extends Number> {
  int f(X x) { return x.intValue(); }
}
int g(A<String> y) { return y.f("hello"); }
```

The function g seems unsafe since any of its execution resolves in the impossible computation of "hello".intValue(). In FGJ$_\omega$, the method g cannot be defined and thus will never be executed because types like A<String> where some arguments do not conform to the bounds of the corresponding parameters are not well-formed. However, in order to prevent any execution of g, it is sufficient to prevent the creation of values of such types. Indeed, if no such value can ever be constructed, it will never be possible to call the seemingly unsafe method g. That is exactly what FGJ$_\Omega$ does; types like A<String> are well-formed but no values of that type can ever be constructed. Thus, the method g, which is well-formed in FGJ$_\Omega$, will never be executed.

Note that in FGJ$_\omega$, like in FGJ$_\Omega$, values of types like A<String> can never be constructed. Thus, the type system of FGJ$_\omega$ is strictly more constraining than the one of FGJ$_\Omega$. The advantage of the additional constraints is that they prevent the definition of methods like g, which would anyway never be executed even if they were allowed. The disadvantage is that they significantly increase the complexity of a type safety proof based on the proofs of subjection reduction and progress even though they are not needed at all to prove the type safety.

Erased bounds

In the rule SAT, which verifies that in an environment Δ a type constructor K conforms to a parameter $X\texttt{<}\overline{P}\texttt{>} \triangleleft C\texttt{<}\overline{K}\texttt{>}^?$, the premise that verifies that the constructor K conforms to the bound $C\texttt{<}\overline{K}\texttt{>}$ uses the environment Δ augmented with the parameters $\overline{P}$ but with their bounds erased. Intuitively, the bounds should be kept and this would probably be sound. This restriction was introduced for technical reasons; it lets us prove that the subtyping judgment is preserved by substitution of type constructors $\overline{K}$ (appendix, Lemma 22). With erased bounds, this important property can be verified using a simple induction on the structural size of the kind of the substituted type constructors $\overline{K}$ followed by an induction on the derivation depth of the subtyping judgment. The restriction allows for a simpler proof but, as illustrated below, it has for consequence that less programs are well-formed. However, we believe that it has a very limited impact. For instance none of our examples is affected by this restriction and we are not aware of interesting examples where the more general rule would be necessary.

```
class A<F<X extends Number> extends Number> { }
new A<<Y> => Y>()
```

In order for the program to be well-formed, the type constructor `<Y> => Y` has to conform to the parameter `F<X extends Number> extends Number`. Therefore, the rule SAT requires that `<Y> => Y` is applicable to any possible argument of `F`, which is trivially true since `Y` has no bound, and that in an environment consisting of the sole variable `X`, `(<Y> => Y)<X>`, which reduces to `X`, is a subtype of `Number`, which is true only if the variable `X` occurs in the environment with its bound `Number`. As the rule SAT erases this bound, the above program is rejected.

The rule SAT is the only rule where bounds are erased. For instance the bounds of a method's type parameters are used to typecheck the body of the method. This is the case for the following method which is well-formed only because `X` is declared as a subtype of `Number`.

```
<X extends Number> Number foo(X x) { return x; }
```

5 RELATED WORK

Higher-order subtyping The combination of subtyping and higher-order polymorphism that exists in FGJ_ω has been intensively studied in the context of lambda-calculi ($F^\omega_{<:}$ [PS94], $F^\omega_{\leq}$ [CG03]) and object-calculi ($Ob_{\omega<:\mu}$ [AC96]) under the name of *higher-order subtyping*. In FGJ_ω, subtyping is restricted to the comparison of types while, with higher-order subtyping, subtyping is lifted to arbitrary type constructors using a pointwise comparison. In our examples, we never had to compare type constructors for subtyping but we admit that this feature could simplify an extension of our design with variant type constructors. We believe that our proof of type safety could be adapted but we provide no evidence of it.

The originality of our work is to consider a calculus that is close to a real object-oriented programming language so that no encoding is needed, neither for objects nor for classes. Our set of typing rules can appear larger and more complex than those of the previously cited calculi but it is self-contained: in order to reach the same level of formalization of a class-based language, the cited calculi should also include complex encodings of objects, classes, inheritance, etc.

Contrary to $F^\omega_{<:}$ and $Ob_{\omega<:\mu}$, our calculus has bounds on type constructor abstractions, so that we can write `<X extends String> => List<X>` and not simply `<X> => List<X>`. This feature, called bounded type operator abstraction, is already present in $F^\omega_{\leq}$ but FGJ_ω additionally allows the parameters to mutually depend on each other in their bounds, a feature that is called F-bounded polymorphism [CCH+89]. It is used by our solution for expressing binary methods of generic data-types in Section 1. FGJ_ω is, to our knowledge, the first calculus to provide F-bounds in combination with type constructor parameterization.

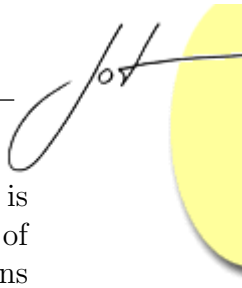
Like the previously cited calculi, FGJ$_\omega$ does not identify eta-convertible type constructors like `List` and `<X> => List<X>`. The responsibility of writing types in such a manner that the compiler never has to compare such type constructors is left to the programmer.

Monads The example presented in Section 1 is similar to what Haskell programmers call monads [Wad92], an abstraction for pluggable computations. One peculiarity of our implementation is that the "plug" operation (`flatMap`) is a method of the class representing the monadic data-type (`ICollection`), while it is an external function (written >>= and called "bind") in Haskell.

GADTs In Section 2, our encoding of GADTs relies on our extension of Java with type constructor parameterization. There exists a similar solution [KR05] that is based on an extension of Java with type equality constraints.

Virtual types Scala virtual types [Ot07] are able to express some use cases of type constructor parameterization [ACb, MPJ06]. The principle is to encode the declaration of a parameter representing a type constructor like `X<_>` as the declaration of an abstract type (type parameter or virtual type) `X` bounded by `Arity1`, where `Arity1` is a class with a single virtual type (say `A1`). A type like `X<String>` can then be expressed in Scala by the *refined type* `X{type A1 = String}`, a sort of intersection type which represents all instances of `X` in which the virtual type `A1` is equal to `String`. It is an open question whether FGJ$_\omega$ can be entirely encoded this way, especially when more complex bounds and anonymous type constructors are involved.

Implementation Shortly after we implemented and made public a prototype compiler for FGJ$_\omega$, type constructor parameterization was independently integrated into the Scala compiler under the name of type constructor polymorphism [MPO07]. The authors of the Scala implementation based their syntax on an unpublished version of the present paper where type constructor parameterization were described in the context of Scala. However they extended our syntax to make type constructor parameterization smoothly interact with pre-existing features of the language. In particular, the Scala implementation provides declaration-site variance annotation for type constructor parameters and adds the concept of class members representing type constructors. This last feature is an extension of the mechanism of virtual types in Scala, it permits abstract declarations, like `type T[X] <: Pair[X,X]`, and concrete ones like `type T[X] = Pair[X,X]`. The latter can be used to emulate FGJ$_\omega$'s anonymous type constructors, which are still missing in Scala. As for FGJ$_\omega$, the problem of inferring type constructor arguments for methods and classes is not addressed by the Scala implementation (such arguments must be explicitly given by the programmer). Our work can be considered as a theoretical foundation for the subset of Scala that corresponds to FGJ$_\omega$.



Anonymous type constructors in Haskell Contrary to our calculus, there is no syntax for anonymous type constructors in Haskell. However a restricted form of anonymous type constructors exists internally; they arise from partial applications of type constructors. For instance, Haskell permits types like `Pair A` even if `Pair` has been declared with two type parameters. Such a construct is actually a partial type application and is equivalent to `<X> => Pair<A,X>` in our design. Thus, Haskell is able to represent all the anonymous type constructors that correspond to partial type applications but not the others. This is a restriction compared to our design: for example, the FGJ$_\omega$ anonymous type constructor `<X> => Pair<X,A>` is not expressible as a partial application of `Pair`. In Haskell, which has a complete type inference mechanism based on unification, such a limitation is needed to ensure decidability of type-checking. Indeed the unification between types that contain anonymous type constructors (also called higher-order unification) is a well-known undecidable problem. In FGJ$_\omega$ we adopt the Java philosophy of explicit type annotations along with some simple type inference (in fact none in our prototype) and are therefore not bound by this limitation. A study of an extension of Haskell with anonymous functions at the level of types [NT02] shows there is some interest for this feature even in the Haskell community.

6 CONCLUSION

Summary

This paper describes a simple way of adding type constructor parameterization to Java. The main elements of the proposed extension, a generalized syntax for type parameters and generalized typing rules, have been gradually introduced through two concrete examples that cannot be expressed with type parameterization only. Our extension, called FGJ$_\omega$, has been implemented in a prototype compiler, formalized and important properties of its type system, like decidability and safety, have been proven. To our knowledge, our work is the first to propose a design (that is proven sound) for integrating type constructor parameterization to Java. Our work constitutes also the first proof of safety for a type system that combines higher-order polymorphism and F-bounded polymorphism. The degree of confidence of this proof is very high since most parts of it, including all the difficult ones, have been mechanically checked in a theorem prover. Our proof is based on an original technique that breaks the problematic dependency between type well-formedness and subtyping by considering a more general type system, FGJ$_\Omega$ A similar technique could be used to simplify the type safety proof of FGJ.

Future work

Our calculus still misses some useful features of Java-like languages, for instance wildcard types [IV06, THE$^+$04], which have been popularized by their implementa-

tion in Java. A continuation of this work would be to study, at a theoretical level, the interaction between type constructor parameters and wildcard types, in particular with respect to the decidability of subtyping [KP06]. From a more pragmatic point of view, it would also be interesting to investigate whether type constructor parameters are compatible with the most common strategies for type argument inference in Java-like languages [OZZ01, Ode02].

Acknowledgments We would like to thank the anonymous referees, whose insightful and very detailed comments helped us significantly improve the paper. We also thank Rachele Fuzzati for helping us improving the explanations.

REFERENCES

[ACa] Philippe Altherr and Vincent Cremet. FGJ-omega web page. http://lamp.epfl.ch/∼cremet/FGJ-omega/.

[ACb] Philippe Altherr and Vincent Cremet. Messages posted on the Scala mailing list. Accessible from http://lamp.epfl.ch/∼cremet/FGJ-omega/.

[AC96] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer Verlag, 1996.

[BBLS05] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82, 2005. Special issue.

[BCC+96] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[CCH+89] Peter Canning, William Cook, Walter Hill, Walter Olthoff, and John C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA'89)*, pages 273–280, New York, NY, USA, 1989. ACM Press.

[CG03] Adriana B. Compagnoni and Healfdene H. Goguen. Typed operational semantics for higher order subtyping. *Information and Computation*, 184:242–297, August 2003.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[IPW99] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 1999. Full version in ACM Transactions on Programming Languages and Systems (TOPLAS), 23(3), May 2001.

[IV06] Atsushi Igarashi and Mirko Viroli. Variant parametric types: A flexible subtyping scheme for generics. *ACM Transactions on Programming Languages and Systems*, 28(5):795–847, 2006.

[Jon03] Simon Peyton Jones. The Haskell 98 language and libraries: The revised report. Cambridge University Press, 2003.

[KP06] Andrew J. Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance, September 2006. FOOL-WOOD '07.

[KR05] Andrew Kennedy and Claudio Russo. Generalized algebraic data types and object-oriented programming. In *ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. ACM Press, October 2005.

[MPJ06] Adriaan Moors, Frank Piessens, and Wouter Joosen. An object-oriented approach to datatype-generic programming. In *Workshop on Generic Programming (WGP'2006)*. ACM, September 2006.

[MPO07] Adriaan Moors, Frank Piessens, and Martin Odersky. Towards equal rights for higher-kinded types. Accepted for the 6th International Workshop on Multiparadigm Programming with Object-Oriented Languages at the European Conference on Object-Oriented Programming (ECOOP), 2007.

[NT02] Matthias Neubauer and Peter Thiemann. Type classes with more higher-order polymorphism. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02)*, pages 179–190, New York, NY, USA, 2002. ACM Press.

[Ode02] Martin Odersky. Inferred type instantiation for GJ. Note sent to the types mailing list, January 2002.

[Ot07] Martin Odersky and the Scala Team. The Scala Language Specification (version 2.5). http://www.scala-lang.org/docu/files/ScalaReference.pdf, June 2007.

[OZZ01] Martin Odersky, Matthias Zenger, and Christoph Zenger. Colored local type inference. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 41–53, 2001.

[Pro04] The Coq Development Team (LogiCal Project). The Coq proof assistant reference manual (version 8.0). http://coq.inria.fr, 2004.

[PS94] Benjamin C. Pierce and Martin Steffen. Higher-order subtyping. In *IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, 1994. Full version in *Theoretical Computer Science*, vol. 176, no. 1–2, pp. 235–282, 1997 (corrigendum in TCS vol. 184 (1997), p. 247).

[Tai67] William W. Tait. Intensional interpretations of functionals of finite type I. *Journal of Symbolic Logic*, 32(2):198–212, June 1967.

[THE+04] Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ah, Gilad Bracha, and Neal Gafter. Adding wildcards to the Java programming language. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 1289–1296. ACM Press, 2004.

[Wad92] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences*. Springer-Verlag, August 1992. Also in J. Jeuring and E. Meijer, editors, Advanced Functional Programming, Springer Verlag, LNCS 925, 1995.

[WF94] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115, 1994.

ABOUT THE AUTHORS

Vincent Cremet received a Ph.D. in Computer Science from EPFL, Switzerland in 2006. His research interests include the design and formal proof of advanced type systems for object-oriented languages. He can be reached at vincent.cremet@gmail.com. See also http://lamp.epfl.ch/∼cremet/.

Philippe Altherr received a Ph.D. in Computer Science from EPFL, Switzerland in 2006. His research interests include programming language design and compiler implementation techniques. His email address is philippe.altherr@gmail.com.

APPENDIX INTRODUCTION

In this appendix we present a formal description of our two calculi, FGJ_ω and FGJ_Ω. Starting from such description we then prove that the type systems of both FGJ_ω and FGJ_Ω are safe and that FGJ_ω's type system is also decidable.

Well-kindedness vs. well-formedness

The complexity of the type safety proof is caused by the dependency of type well-formedness on subtyping. This dependency arises from the fact that type parameters have bounds and that arguments in type applications must be subtypes of these bounds. For example, if a class `C` has a single parameter `X extends String`, the type `C<`T`>` is *well-formed*, only if T is a *subtype* of `String`. The dependency is already present in FGJ but it becomes really problematic only in the context of FGJ_ω, where types are potentially reducible. In particular, the proof of the preservation of type well-formedness by reduction is difficult.

Our solution to make the proof of type safety tractable is to define a new notion of type well-formedness that does not enforce the above mentioned subtyping constraints. Types that are well-formed with respect to this new definition are called *well-kinded*; they satisfy the minimal property that type constructors are always applied to arguments of the right arity. Types that are well-formed with respect to the classical definition are simply called *well-formed*.

Implementation in Coq

The type safety proof of FGJ_Ω presented in this appendix has been formalized in the Coq proof assistant. The implementation of the proof in Coq is available on the FGJ_ω home page [ACa]. In Coq we have represented binders by De Bruijn's indices instead of variables and, to simplify the proof, we have not considered methods overriding inherited concrete methods since they do not play a central role in the complexity of the proof.

Appendix overview

Section A defines FGJ_ω. Section B proves some basic properties of well-kinded types. Section C proves that the type system of FGJ_ω is decidable. Section D defines FGJ_Ω and proves that its type system is more general than the one of FGJ_ω. Section E proves that FGJ_Ω is type safe. Since FGJ_Ω is a relaxed version of FGJ_ω, this trivially implies that FGJ_ω is also type safe.

A DEFINITION OF FGJ$_\omega$

Preamble: first-class rows

In the complete formalization of FGJ$_\omega$ we refine the syntax presented in Section 3 by grouping type constructors into tuples (called *rows* in the following). Before justifying this choice, we describe the changes that such refinement introduces in the syntax. We consider the example of a generic class for representing pairs expressed in the original syntax.

```
class Pair<X0<>,X1<>> { X0<> fst; X1<> snd;
  Pair<X1,X0> reverse() { return new Pair<X1,X0>(snd, fst); }
}
```

In the code that follows, we have applied the following transformations. The two type constructor parameters `X0` and `X1` are replaced with a single row parameter `X`. The description of its two components are put in parentheses after the delimiter `::`. Every occurrence of `X0` in the class is replaced with a row projection `X@0` that denotes the first component of the row `X` (and similarly for `X1`). The syntax `(X@1,X@0)` denotes a concrete row made of the two type constructors `X@1` and `X@0`. Such a row is passed as argument to `Pair` inside the method `reverse`.

```
class Pair<X :: (<>,<>)> { X@0<> fst; X@1<> snd;
  Pair<(X@1,X@0)> reverse() { return new Pair<(X@1,X@0)>(snd, fst); }
}
```

As we can see, the correspondence between both syntaxes is quite direct. The changes in the syntax are summarized in the following table.

Original syntax			Syntax with first-class rows		
Π	::=	$\overline{P}$	Π	::=	$X :: (\overline{P})$
P	::=	X<Π> $\triangleleft$ $N^?$	P	::=	<Π> $\triangleleft$ $N^?$
K	::=	X | C | <Π> => T	K	::=	$R@i$ | C | <Π> => T
T	::=	K<$\overline{K}$>	T	::=	K<R>
N	::=	C<$\overline{K}$>	N	::=	C<R>
			R	::=	X | $(\overline{K})$

In FGJ$_\omega$, and also in FGJ, it is natural to think of the type parameters that are declared by a same class or a same method as a group since they can be mutually recursive in their bounds. In particular, they must be *simultaneously* substituted for some corresponding arguments. Note that such simultaneous substitutions are undefined if not enough arguments are provided. For example, what does the variable X_1 become in the simultaneous substitution `Pair<`X_0`,`X_1`>[(`X_0`,`X_1`)\(String)]`?

Having substitutions that are potentially undefined requires to formally treat them as partial functions, which is quickly intractable in a proof. Actually, we could prove that well-kinded types never lead to undefined substitutions but then, every single lemma should be augmented with the hypothesis that every involved type is well-kinded, which is incredibly heavy.

Making rows explicit in the syntax solves the problem posed by undefined simultaneous substitutions since a simultaneous substitution is decomposed into the single substitution of a row, which is always defined, and the reduction of one or several type projections. The previously undefined substitution becomes well-defined when explicit rows are used. Admittedly, the resulting type gets stuck (because `(String)@1` is not reducible) but this is a kind of problem that is not technically difficult to solve.

$$\begin{array}{ll} & \texttt{Pair<}(X\texttt{@}0, X\texttt{@}1)\texttt{>}[X \backslash \texttt{(String)}] \\ = & \texttt{Pair<}((\texttt{String})\texttt{@}0, (\texttt{String})\texttt{@}1)\texttt{>} \\ \rightarrow & \texttt{Pair<}(\texttt{String}, (\texttt{String})\texttt{@}1)\texttt{>} \end{array}$$

Syntax of FGJ$_\omega$

The abstract syntax of FGJ$_\omega$ is summarized in Fig. 1.

Semantics of FGJ$_\omega$

The semantics of FGJ$_\omega$ is defined in Fig. 3. It comes as a one-step reduction relation that must be repeatedly applied in order to reach a value. The definition uses auxiliary functions defined in Fig. 2 for, respectively, collecting the field names of a class ($\text{fields}(C) = \overline{f}$), finding the class that contains the implementation of a given method ($\text{lookup}_C(m) = C'$) and computing the type instantiation $C\texttt{<}R\texttt{>}$ of a class C as seen from a class type N ($\text{superclass}_C(N) = C\texttt{<}R\texttt{>}$).

Typing rules of FGJ$_\omega$

The typing rules use the following typing entities.

Typing entities				
Kind	k	$::=$	$*$	type kind
		$\mid$	$k \rightarrow *$	type constructor kind
		$\mid$	$(\overline{k})$	row kind
Type environment	Δ	$::=$	$\emptyset \mid \Delta, \Pi$	
Term environment	Γ	$::=$	$\emptyset \mid \Gamma, x\colon T$	

Typing rules of FGJ$_\omega$ are described by several judgments. The following table presents the main judgments with their interpretation.

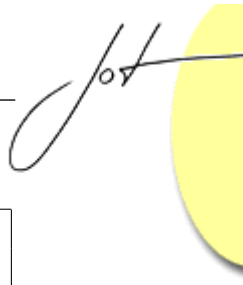
Symbols				
Class name	C	::=	$\cdots$	
Method name	m	::=	$\cdots$	
Field name	f	::=	$\cdots$	
Variable	x	::=	`this` $\mid$ $\cdots$	
Row variable	X, Y	::=	$\cdots$	
Programs				
Program	p	::=	$\overline{D}$ `return` t;	
Class declaration	D	::=	`class` C<Π> $\triangleleft$ $N^?$ { $\overline{F}$ $\overline{M}$ }	
Field declaration	F	::=	$T\ f$;	
Method declaration	M	::=	<Π> $T\ m(\overline{T}\ \overline{x})$ { `return` t; }	
Term	t, u	::=	x	method parameter, current instance
		$\mid$	$t.f$	field selection
		$\mid$	$t.$<R>$m(\overline{t})$	method call
		$\mid$	`new` $N(\overline{t})$	instance creation
Row parameter	Π	::=	$X :: (\overline{P})$	
Type constr. descriptor	P, Q	::=	<Π> $\triangleleft$ $N^?$	
Constructor row	R	::=	X	row variable
		$\mid$	$(\overline{K})$	row constructor
Type constructor	K	::=	$R@i$	row projection
		$\mid$	C	class type constructor
		$\mid$	<Π> => T	anonymous type constructor
Type	S, T, U	::=	K<R>	
Class type	N	::=	C<R>	

Figure 1: Syntax of FGJ$_\omega$

$\Delta \vdash T :: k$	T is well-kinded (of kind k)	Fig. 5
$T \to T'$	T reduces in one-step to T'	Fig. 6
$\Delta \vdash T <: U$	T is a subtype of U	Fig. 7
$\Delta \vdash K \in P$	K conforms to P	Fig. 8
$\Delta \vdash R \in \Pi$	R conforms to Π	Fig. 8
$\Delta \vdash T ::: k$	T is well-formed (of kind k)	Fig. 9
$\Delta; \Gamma \vdash t : T$	t has type T	Fig. 10
override$(m, C$<Π>$, C')$	C overrides the method m of class C'	Fig. 11
C<Π> $\vdash F(,\text{resp. } M)$ WF	F (resp. M) is well-formed in C	Fig. 11
$\vdash D$ WF	D is well-formed	Fig. 11
$\vdash p$ WF	p is well-formed	Fig. 11

In the rules, $T[X \backslash R]$ stands for the substitution of the row R for the variable X in T, and $t[x \backslash v]$ stands for the substitution of the value v for the variable x in t.

The reduction from a type construct A to a type construct A' (Fig. 6) is noted $A \to A'$. The reflexive transitive closure of this relation is noted $A \xrightarrow{*} A'$.



$$\frac{\texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd \texttt{none}\ \{\,\overline{T}\,\overline{f};\ \overline{M}\,\}}{\mathrm{fields}(C) = \overline{f}} \qquad \frac{\texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd C'\texttt{<}R\texttt{>}\ \{\,\overline{T}\,\overline{f};\ \overline{M}\,\} \quad \mathrm{fields}(C') = \overline{f'}}{\mathrm{fields}(C) = \overline{f'}, \overline{f}}$$

$$\frac{m \text{ is defined in } C}{\mathrm{lookup}_C(m) = C} \qquad \frac{m \text{ is not defined in } C \quad \texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd C'\texttt{<}R\texttt{>}\ \{\,\overline{F}\ \overline{M}\,\} \quad \mathrm{lookup}_{C'}(m) = C''}{\mathrm{lookup}_C(m) = C''}$$

$$\frac{}{\mathrm{superclass}_C(C\texttt{<}R\texttt{>}) = C\texttt{<}R\texttt{>}} \qquad \frac{\texttt{class}\ C\texttt{<}X :: (\overline{P})\texttt{>} \lhd N\ \{\,\overline{F}\ \overline{M}\,\} \quad C \neq C' \quad \mathrm{superclass}_{C'}(N[X\backslash R]) = C'\texttt{<}R'\texttt{>}}{\mathrm{superclass}_{C'}(C\texttt{<}R\texttt{>}) = C'\texttt{<}R'\texttt{>}}$$

$$\frac{\texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd N^{?}\ \{\,\overline{F}\ \overline{M}\,\} \quad \Pi = X :: (\overline{P})}{F_i \in C\texttt{<}X\texttt{>}} \qquad \frac{\texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd N^{?}\ \{\,\overline{F}\ \overline{M}\,\} \quad \Pi = X :: (\overline{P})}{M_i \in C\texttt{<}X\texttt{>}}$$

Figure 2: Fields ($\mathrm{fields}(C) = \overline{f}$), method lookup ($\mathrm{lookup}_C(m) = C'$), superclass instantiation ($\mathrm{superclass}_C(N) = C\texttt{<}R\texttt{>}$) and member access ($F, M \in C\texttt{<}X\texttt{>}$)

B PROPERTIES OF WELL-KINDED TYPES

This section enumerates a set of basic properties about well-kinded types. For readability, properties are sometimes represented as admissible inference rules. The proofs are similar in structure and complexity to the corresponding proofs for the simply typed λ-calculus.

Lemma 1 [Type well-kindedness is preserved by type substitution]

Let $\Pi = X :: (\overline{P})$. Assume $\Delta \vdash R :: \mathrm{kind}(\Pi)$.

$$\frac{\Delta, \Pi, \Delta' \vdash T :: *}{\Delta, \Delta'[X\backslash R] \vdash T[X\backslash R] :: *}$$

$$\text{(R-Context)}\ \frac{t \rightarrow t'}{e[t] \rightarrow e[t']} \qquad \text{(R-Select)}\ \frac{v = \texttt{new}\ C\texttt{<}R\texttt{>}(\overline{v}) \qquad \text{fields}(C) = \overline{f}}{v.f_i \rightarrow v_i}$$

$$\text{(R-Call)}\ \frac{\begin{array}{c} v = \texttt{new}\ C\texttt{<}R\texttt{>}(\overline{w}) \qquad C' = \text{lookup}_C(m) \\ \texttt{<}Y :: (\overline{P})\texttt{>}\ T\ m(\overline{T}\ \overline{x})\ \{\ \texttt{return}\ t;\ \} \in C'\texttt{<}X\texttt{>} \\ C'\texttt{<}R'\texttt{>} = \text{superclass}_{C'}(C\texttt{<}R\texttt{>}) \end{array}}{v.\texttt{<}R''\texttt{>}m(\overline{v}) \rightarrow t[X \backslash R'][Y \backslash R''][\texttt{this} \backslash v][\overline{x} \backslash \overline{v}]}$$

Semantic entities

Value	v, w	::=	$\texttt{new}\ N(\overline{v})$
Evaluation context	e	::=	$[\,] \mid e.f \mid e.\texttt{<}R\texttt{>}m(\overline{t}) \mid v.\texttt{<}R\texttt{>}m(\overline{v}, e, \overline{t})$
		$\mid$	$\texttt{new}\ N(\overline{v}, e, \overline{t})$

Figure 3: One-step reduction ($t \rightarrow u$)

$$\frac{R \text{ irreducible}}{\text{bound}_\Delta(C\texttt{<}R\texttt{>}) = C\texttt{<}R\texttt{>}}$$

$$\frac{R \text{ irreducible} \qquad (X :: (\overline{P})) \in \Delta \qquad P_i = \texttt{<}Y :: (\overline{Q})\texttt{>} \triangleleft N}{\text{bound}_\Delta(X@i\texttt{<}R\texttt{>}) = N[Y \backslash R]}$$

$$\frac{T \rightarrow T' \text{ (leftmost reduction)} \qquad \text{bound}_\Delta(T') = N}{\text{bound}_\Delta(T) = N}$$

$$\frac{\texttt{class}\ C\texttt{<}\Pi\texttt{>} \triangleleft N^? \ \{\ \overline{F}\ \overline{M}\ \}}{\text{params}_\Delta(C) = \Pi}$$

$$\frac{}{\text{params}_\Delta(\texttt{<}\Pi\texttt{>}\ \texttt{=>}\ T) = \Pi}$$

$$\frac{(X :: (\overline{P})) \in \Delta \qquad P_i = \texttt{<}\Pi\texttt{>} \triangleleft N^?}{\text{params}_\Delta(X@i) = \Pi}$$

$$\frac{\text{params}_\Delta(K_i) = \Pi}{\text{params}_\Delta((\overline{K})@i) = \Pi}$$

Figure 4: Bound ($\text{bound}_\Delta(T) = N$) and parameters ($\text{params}_\Delta(K) = \Pi$)

Proof: Similar to the preservation of typing by substitution in the simply typed lambda-calculus. **Qed**

Lemma 2 [Type well-kindedness is preserved by type reduction]

$$\text{(WK-Var)}\ \frac{(X :: (\overline{P})) \in \Delta}{\Delta \vdash X :: (\text{kind}(\overline{P}))} \qquad \text{(WK-Row)}\ \frac{\Delta \vdash \overline{K} :: \overline{k}}{\Delta \vdash (\overline{K}) :: (\overline{k})}$$

$$\text{(WK-Class)}\ \frac{\texttt{class}\ C\texttt{<}\Pi\texttt{>} \triangleleft N^{?}\ \{\,\overline{F}\ \overline{M}\,\}}{\Delta \vdash C :: \text{kind}(\Pi) \to *} \qquad \text{(WK-Proj)}\ \frac{\Delta \vdash R :: (\overline{k})}{\Delta \vdash R@i :: k_i}$$

$$\text{(WK-Fun)}\ \frac{\Delta \vdash \Pi\ \text{WK} \quad \Delta, \Pi \vdash T :: *}{\Delta \vdash \texttt{<}\Pi\texttt{>} \Rightarrow T :: \text{kind}(\Pi) \to *} \qquad \text{(WK-Apply)}\ \frac{\Delta \vdash K :: k \to * \quad \Delta \vdash R :: k}{\Delta \vdash K\texttt{<}R\texttt{>} :: *}$$

$$\text{(WK-Param)}\ \frac{\Delta \vdash \Pi\ \text{WK} \quad (\Delta, \Pi \vdash N :: *)^{?}}{\Delta \vdash \texttt{<}\Pi\texttt{>} \triangleleft N^{?}\ \text{WK}} \qquad \text{(WK-Section)}\ \frac{\Delta, X :: (\overline{P}) \vdash \overline{P}\ \text{WK}}{\Delta \vdash (X :: (\overline{P}))\ \text{WK}}$$

$$\text{where}\ \begin{cases} \text{kind}(\Pi) = (\text{kind}(\overline{P})) & \text{if } \Pi = X :: (\overline{P}) \\ \text{kind}(P) = \text{kind}(\Pi) \to * & \text{if } P = \texttt{<}\Pi\texttt{>} \triangleleft N^{?} \end{cases}$$

Figure 5: Type well-kindedness ($\Delta \vdash K, T, R :: k$, $\Delta \vdash P, \Pi$ WK). Kinds ($\text{kind}(\Pi) = k, \text{kind}(P) = k$)

$$\text{(RT-App)}\ \frac{K = \texttt{<}X :: (\overline{P})\texttt{>} \Rightarrow T}{K\texttt{<}R\texttt{>} \to T[X \backslash R]} \qquad \text{(RT-Proj)}\ \frac{}{(\overline{K})@i \to K_i} \qquad \text{(RT-Ctx)}\ \frac{A \to A'}{E[A] \to E[A']}$$

$$\text{where}\ \begin{cases} \text{Type construct} & A & ::= & T \mid K \mid R \mid \Pi \mid P \\ \text{Red. context} & E & ::= & [\,] \mid E\texttt{<}R\texttt{>} \mid K\texttt{<}E\texttt{>} \mid E@i \mid \texttt{<}E\texttt{>} \Rightarrow T \\ & & \mid & \texttt{<}\Pi\texttt{>} \Rightarrow E \mid (\overline{K}, E, \overline{K}') \\ & & \mid & \texttt{<}E\texttt{>} \triangleleft N^{?} \mid \texttt{<}\Pi\texttt{>} \triangleleft E \mid X :: (\overline{P}, E, \overline{Q}) \end{cases}$$

Figure 6: Reduction of type constructs ($A \to A'$)

$$\frac{\Delta \vdash T\ \text{WK} \qquad T \to U}{\Delta \vdash U\ \text{WK}}$$

$$\text{(S-Refl)}\ \frac{}{\Delta \vdash T <: T}$$

$$\text{(S-Class)}\ \frac{\texttt{class}\ C\texttt{<}X :: (\overline{P})\texttt{>} \triangleleft N\ \{\,\overline{F}\ \overline{M}\,\} \quad \Delta \vdash N[X\backslash R] <: U}{\Delta \vdash C\texttt{<}R\texttt{>} <: U} \qquad \text{(S-Var)}\ \frac{(X :: (\overline{P})) \in \Delta \quad P_i = \texttt{<}Y :: (\overline{Q})\texttt{>} \triangleleft N \quad \Delta \vdash N[Y\backslash R] <: U}{\Delta \vdash X@i\texttt{<}R\texttt{>} <: U}$$

$$\text{(S-Red}_\text{L}\text{)}\ \frac{T \to T' \quad \Delta \vdash T' <: U}{\Delta \vdash T <: U} \qquad \text{(S-Red}_\text{R}\text{)}\ \frac{U \to U' \quad \Delta \vdash T <: U'}{\Delta \vdash T <: U}$$

Figure 7: Subtyping ($\Delta \vdash T <: U$)

$$\text{(Poly-Sat)}\ \frac{\forall i,\ \Delta \vdash R@i \in P_i[X\backslash R]}{\Delta \vdash R \in (X :: (\overline{P}))}$$

$$\text{(Sat)}\ \frac{\Pi = X :: (\overline{P}) \qquad \Pi' = \text{params}_\Delta(K) \qquad \Delta, \Pi \vdash X \in \Pi' \quad \Pi_0 = \text{erase}(\Pi) \qquad (\Delta, \Pi_0 \vdash K\texttt{<}X\texttt{>} <: N)^?}{\Delta \vdash K \in \texttt{<}\Pi\texttt{>} \triangleleft N^?}$$

$$\text{where}\ \begin{cases} \text{erase}(X :: (\overline{P})) & = & X :: (\text{erase}(\overline{P})) \\ \text{erase}(\texttt{<}\Pi\texttt{>} \triangleleft N^?) & = & \texttt{<}\text{erase}(\Pi)\texttt{>} \triangleleft \texttt{none} \end{cases}$$

Figure 8: Satisfaction ($\Delta \vdash R \in \Pi$, $\Delta \vdash K \in P$), and bound erasure ($\text{erase}(\Pi) = \Pi'$, $\text{erase}(P) = P'$)

Proof: Similar to the preservation of typing by reduction in the simply typed lambda-calculus. Using Lemma 1. **Qed**

Lemma 3 [Confluence of type reduction]

$$\frac{T \xrightarrow{*} S_1 \qquad T \xrightarrow{*} S_2}{\exists U,\quad S_1 \xrightarrow{*} U \ \wedge\ S_2 \xrightarrow{*} U}$$

$$\text{(WF-Var)}\ \frac{(X :: (\overline{P})) \in \Delta}{\Delta \vdash X ::: (\text{kind}(\overline{P}))} \qquad \text{(WF-Row)}\ \frac{\Delta \vdash \overline{K} ::: \overline{k}}{\Delta \vdash (\overline{K}) ::: (\overline{k})}$$

$$\text{(WF-Class)}\ \frac{\texttt{class}\ C\texttt{<}\Pi\texttt{>} \triangleleft N^? \ \{\, \overline{F}\ \overline{M} \,\}}{\Delta \vdash C ::: \text{kind}(\Pi) \to *} \qquad \text{(WF-Proj)}\ \frac{\Delta \vdash R ::: (\overline{k})}{\Delta \vdash R@i ::: k_i}$$

$$\text{(WF-Fun)}\ \frac{\Delta \vdash \Pi\ \text{wf} \quad \Delta, \Pi \vdash T ::: *}{\Delta \vdash \texttt{<}\Pi\texttt{>} \Rightarrow T ::: \text{kind}(\Pi) \to *} \qquad \text{(WF-Apply)}\ \frac{\Delta \vdash K ::: k \to * \quad \Delta \vdash R ::: k \quad \Pi = \text{params}_\Delta(K) \quad \Delta \vdash R \in \Pi}{\Delta \vdash K\texttt{<}R\texttt{>} ::: *}$$

$$\text{(WF-Param)}\ \frac{\Delta \vdash \Pi\ \text{wf} \quad (\Delta, \Pi \vdash N ::: *)^?}{\Delta \vdash \texttt{<}\Pi\texttt{>} \triangleleft N^?\ \text{wf}} \qquad \text{(WF-Section)}\ \frac{\Delta, X :: (\overline{P}) \vdash \overline{P}\ \text{wf}}{\Delta \vdash (X :: (\overline{P}))\ \text{wf}}$$

Figure 9: Type well-formedness ($\Delta \vdash K, T, R ::: k$, $\Delta \vdash P, \Pi$ wf)

Proof: The confluence of type reduction can be proven similarly to the confluence of term reduction in the lambda-calculus, for example using the concept of *parallel reductions*. **Qed**

Lemma 4 [Well-kinded types are strongly normalizable]

The principle of the proof that well-kinded types are strongly normalizable is already explained in Section 4. It relies on a translation from well-kinded types to well-typed terms of the simply typed lambda-calculus with pairs. The detail of the translation and the proofs that the translation preserves typing and reduction can be found in the Coq formalization [ACa].

C DECIDABILITY OF FGJ$_\omega$'S TYPE SYSTEM

Lemma 5 [Well-foundedness of the class inheritance relation is decidable]

Since the number of classes is finite, verifying the absence of cycles in the class inheritance relation is decidable. For a binary relation over a finite set, the absence of cycles is equivalent to well-foundedness. Consequently, the well-foundedness of the class inheritance relation is decidable.

$$\text{(T-Var)}\ \frac{\Gamma = \overline{x}{:}\overline{T}}{\Delta;\ \Gamma \vdash\ x_i : T_i} \qquad \text{(T-Select)}\ \frac{\begin{array}{c} U\ f; \in C\texttt{<}X\texttt{>} \qquad \Delta;\ \Gamma \vdash\ t : T \\ N = \mathrm{bound}_\Delta(T) \qquad C\texttt{<}R\texttt{>} = \mathrm{superclass}_C(N) \end{array}}{\Delta;\ \Gamma \vdash\ t.f : U[X\backslash R]}$$

$$\text{(T-Call)}\ \frac{\begin{array}{c} \Delta;\ \Gamma \vdash\ t : T \qquad \Delta \vdash\ R' ::: k \qquad \Delta;\ \Gamma \vdash\ \overline{t} : \overline{T} \\ N = \mathrm{bound}_\Delta(T) \qquad C'\texttt{<}R''\texttt{>} = N \qquad C = \mathrm{lookup}_{C'}(m) \\ C\texttt{<}R\texttt{>} = \mathrm{superclass}_C(N) \\ \texttt{<}Y :: (\overline{P})\texttt{>}\ U\ m(\overline{U}\ \overline{x})\ \{\,\texttt{return}\ u;\,\} \in C\texttt{<}X\texttt{>} \\ \Delta \vdash\ R' \in (Y :: (\overline{P}))[X\backslash R] \qquad \Delta \vdash\ \overline{T} <: \overline{U}[X\backslash R][Y\backslash R'] \end{array}}{\Delta;\ \Gamma \vdash\ t.\texttt{<}R'\texttt{>}m(\overline{t}) : U[X\backslash R][Y\backslash R']}$$

$$\text{(T-New)}\ \frac{\Delta \vdash\ N ::: * \qquad N = C\texttt{<}R\texttt{>} \qquad \overline{f} = \mathrm{fields}(C) \qquad \Delta;\Gamma;N \vdash\ \overline{f} = \overline{t}\ \textsc{wf}}{\Delta;\ \Gamma \vdash\ \texttt{new}\ N(\overline{t}) : N}$$

$$\text{(WF-Field-Impl)}\ \frac{\begin{array}{c} T\ f; \in C\texttt{<}X\texttt{>} \qquad C\texttt{<}R\texttt{>} = \mathrm{superclass}_C(N) \\ \Delta;\ \Gamma \vdash\ t : S \qquad \Delta \vdash\ S <: T[X\backslash R] \end{array}}{\Delta;\Gamma;N \vdash\ f = t\ \textsc{wf}}$$

Figure 10: Type assignment ($\Delta;\ \Gamma \vdash\ t : T$) and well-formedness of field implementations ($\Delta;\Gamma;N \vdash\ f = t\ \textsc{wf}$)

Lemma 6 [Decidability of type well-kindedness]

The type well-kindedness judgment (Fig. 5) is decidable since, for every rule, the types that appear in the premises are smaller, with respect to their structural size, than the types that appear in the conclusion.

Lemma 7 [Preliminary phase]

The well-formedness judgment for programs (rule WF-Program) contains a preliminary phase that checks the conformance of arities (well-kindedness) in the declared type parameter Π and the optional superclass N of each class (but not in its members). This phase is modeled by the premise $\vdash\ \overline{D}\ \textsc{wk}$ and the rule WK-Class-Decl. The phase terminates because of Lemma 6. This preliminary check, together with Lemma 1, implies the interesting property that well-kindedness propagates from a class type to its direct supertype.

Lemma 8 [Auxiliary judgments are decidable and deterministic]

$$\text{(WF-Program)}\ \frac{\begin{array}{c}\text{no duplicate fields in the program}\\ \text{no duplicate methods in a class}\\ \{\,(C,C')\ \mid\ \texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd C'\texttt{<}R\texttt{>}\ \{\,\overline{F}\ \overline{M}\,\}\,\}\ \text{is well-founded}\\ \vdash\ \overline{D}\ \text{WK} \qquad \vdash\ \overline{D}\ \text{WF} \qquad \emptyset;\emptyset \vdash\ t : T\end{array}}{\vdash\ \overline{D}\ \texttt{return}\ t;\ \text{WF}}$$

$$\text{(WK-Class-Decl)}\ \frac{\emptyset \vdash\ \Pi\ \text{WK} \qquad (\Pi \vdash\ N :: *)^{?}}{\vdash\ \texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd N^{?}\ \{\,\overline{F}\ \overline{M}\,\}\ \text{WK}}$$

$$\text{(WF-Class-Decl)}\ \frac{\emptyset \vdash\ \Pi\ \text{WF} \qquad (\Pi \vdash\ N ::: *)^{?} \qquad C\texttt{<}\Pi\texttt{>} \vdash\ \overline{F},\overline{M}\ \text{WF}}{\vdash\ \texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd N^{?}\ \{\,\overline{F}\ \overline{M}\,\}\ \text{WF}}$$

$$\text{(WF-Field-Decl)}\ \frac{\Pi \vdash\ T ::: *}{C\texttt{<}\Pi\texttt{>} \vdash\ T\ f;\ \text{WF}}$$

$$\text{(WF-Method-Decl)}\ \frac{\begin{array}{c}\Pi \vdash\ \Pi'\ \text{WF}\\ \Delta = \Pi,\Pi' \qquad \Delta \vdash\ \overline{T},T ::: * \qquad \Pi = X :: (\overline{P})\\ \Gamma = \texttt{this}\colon C\texttt{<}X\texttt{>},\overline{x}\colon\overline{T} \qquad \Delta;\Gamma \vdash\ t : S \qquad \Delta \vdash\ S <: T\\ \texttt{class}\ C\texttt{<}\Pi\texttt{>} \lhd C'\texttt{<}R\texttt{>}\ \{\,\overline{F}\ \overline{M}\,\}\ \wedge\ C'' = \text{lookup}_{C'}(m)\\ \Rightarrow\ \text{override}(m, C\texttt{<}\Pi\texttt{>}, C'')\end{array}}{C\texttt{<}\Pi\texttt{>} \vdash\ \texttt{<}\Pi'\texttt{>}\ T\ m(\overline{T}\ \overline{x})\ \{\,\texttt{return}\ t;\,\}\ \text{WF}}$$

$$\text{(Valid-Override)}\ \frac{\begin{array}{c}\texttt{<}Y :: (\overline{P})\texttt{>}\ T\ m(\overline{T}\ \overline{x})\ \{\,\texttt{return}\ t;\,\} \in C\texttt{<}X\texttt{>}\\ \texttt{<}Y :: (\overline{P}')\texttt{>}\ T'\ m(\overline{T}'\ \overline{x})\ \{\,\texttt{return}\ t';\,\} \in C'\texttt{<}X'\texttt{>}\\ \Pi = X :: (\overline{Q}) \qquad C'\texttt{<}R\texttt{>} = \text{superclass}_{C'}(C\texttt{<}X\texttt{>})\\ \Delta = \Pi, (Y :: (\overline{P}'))[X'\backslash R] \qquad \Delta \vdash\ Y \in (Y :: (\overline{P}))\\ \Delta \vdash\ \overline{T}'[X'\backslash R] <: \overline{T} \qquad \Delta \vdash\ T <: T'[X'\backslash R]\end{array}}{\text{override}(m, C\texttt{<}\Pi\texttt{>}, C')}$$

Figure 11: Well-formedness of programs ($\vdash\ p$ WF), classes ($\vdash\ D$ WK, $\vdash\ D$ WF) and members ($C\texttt{<}\Pi\texttt{>} \vdash\ F, M$ WF); valid method overriding ($\text{override}(m, C\texttt{<}\Pi\texttt{>}, C')$)

Under the assumptions that the class inheritance relation is well-founded (rule WF-Program) and that well-kinded types are strongly normalizable (Lemma 4), the following judgments are easily shown decidable and "deterministic" (given a set of values in the left-hand side of an equality, there exists at most one corresponding value in the right-hand side).

- $\text{fields}(C) = \overline{F}$
- $\text{lookup}_C(m) = C'$
- $\text{superclass}_C(N) = N'$
- $\text{bound}_\Delta(T) = N$
- $\text{params}_\Delta(K) = \Pi$

Definition 1 [Definition: well-kinded environments]

$$\frac{}{\vdash \emptyset \text{ WK}} \qquad \frac{\vdash \Delta \text{ WK} \qquad \Delta \vdash \Pi \text{ WK}}{\vdash \Delta, \Pi \text{ WK}} \qquad \frac{\vdash \Delta \text{ WK} \qquad \Delta \vdash \overline{T} \text{ WK}}{\vdash \Delta; \overline{x}{:}\overline{T} \text{ WK}}$$

Lemma 9 [Definition-theorem: well-founded type expansion]

The concept of *type expansion* is defined through the following rules. It represents the step that lets us go from a class type to its superclass (E-Extends), from a variable type to its upper-bound (E-Bound) and from a type to its reduced (E-Red).

$$\text{(E-Extends)}\ \frac{\texttt{class}\ C\texttt{<}X :: (\overline{P})\texttt{>} \triangleleft N\ \{\,\overline{F}\ \overline{M}\,\}}{\Delta \vdash C\texttt{<}R\texttt{>} \prec N[X \backslash R]}$$

$$\text{(E-Bound)}\ \frac{(X :: (\overline{P})) \in \Delta \qquad P_i = \texttt{<}Y :: (\overline{Q})\texttt{>} \triangleleft N}{\Delta \vdash X@i\texttt{<}R\texttt{>} \prec N[Y \backslash R]} \qquad \text{(E-Red)}\ \frac{T \to U}{\Delta \vdash T \prec U}$$

A useful property of type expansion is that it preserves well-kindedness: $\Delta \vdash T \prec U$ and $\Delta \vdash T :: *$ and $\vdash \Delta \text{ WK}$ implies $\Delta \vdash U :: *$. The case E-Extends relies on the preliminary phase (Lemma 7).

The important property of type expansion is that it is well-founded, i.e. there is no infinite sequence of type expansion steps starting from a well-kinded type. The proof of this property is a corollary of the three following facts, which must be proven in this order.

- A sequence of type expansion steps starting from a well-kinded class type $C\texttt{<}R\texttt{>}$ is necessarily finite. This is a consequence of the well-foundedness of the class inheritance relation (WF-Program) and the strong normalization of well-kinded types (Lemma 4).
- A sequence of type expansion steps starting from a well-kinded variable type $X@i\texttt{<}R\texttt{>}$ is necessarily finite. This is a consequence of the previous fact (because the bound of a type variable is a class type) and the strong normalization of well-kinded types.
- A sequence of type expansion steps starting from an arbitrary type T is necessarily finite. This is a consequence of the two previous facts and the strong normalization of well-kinded types.

Lemma 10 [Definition-theorem: well-founded sequence extension]

Given a binary relation R, we define the sequence extension R_{seq} of R as the binary relation over sequences such that for all i and y:

$(x_1, \ldots, x_i, \ldots, x_n) R_{\mathsf{seq}} (x_1, \ldots, y, \ldots, x_n)$ iff $x_i R y$.

It is easy to show that if R is well-founded, the sequence extension of R is also well-founded. It follows from this remark and from Lemma 9 that the sequence extension of type expansion, noted $\prec_{\mathsf{seq}}$, is well-founded.

Lemma 11 [Decidability of subtyping]

Given two types T and U that are well-kinded in a well-kinded environment Δ, the judgment $\Delta \vdash T <: U$ is decidable. Indeed, in the premises of each subtyping rule (Fig. 7), the sequence of two types that are involved in a subtyping judgment are smaller, w.r.t. $\prec_{\mathsf{seq}}$, than the sequence of two types that occur in the conclusion of the rule. The well-foundedness of $\prec_{\mathsf{seq}}$ (Lemma 10) implies that the rules can be turned into a terminating recursive algorithm.

To illustrate the principle of the proof, we consider the case S-Class. The conclusion of the rule is the judgment $\Delta \vdash C\texttt{<}R\texttt{>} <: U$, which involves the types $C\texttt{<}R\texttt{>}$ and U. The main premise of the rule is the judgment $\Delta \vdash N[X\backslash R] <: U$, which involves the types $N[X\backslash R]$ and U. Since $\Delta \vdash C\texttt{<}R\texttt{>} \prec N[X\backslash R]$ by rule E-Extends, the sequence of types $(N[X\backslash R], U)$ is smaller, w.r.t. $\prec_{\mathsf{seq}}$, than the sequence $(C\texttt{<}R\texttt{>}, U)$.

The rule S-Red$_{\mathsf{L}}$, and similarly S-Red$_{\mathsf{R}}$, is apparently not algorithmic because the meta-variable T', which is used in the premises of the rule to represent one possible result of reducing the type T, does not appear in the conclusion. An algorithm should be able to "guess" the value of T'. Fortunately there is a finite number of possible type reductions, so an algorithm can just try them in order.

Lemma 12 [Decidability of satisfaction]

The termination of the satisfaction judgment (Fig. 8) is best explained by inlining the rule Sat inside the rule Poly-Sat.

$$(\textsc{Poly-Sat})\ \frac{\begin{array}{c}\text{for all } i, \quad \texttt{<}\Pi\texttt{>} \triangleleft N^? = P_i[X\backslash R] \qquad Y :: (\overline{Q}) = \Pi \\ \Pi' = \mathrm{params}_\Delta(R@i) \qquad \Delta, \Pi \vdash Y \in \Pi' \\ \Pi_0 = \mathrm{erase}(\Pi) \qquad (\Delta, \Pi_0 \vdash R@i\texttt{<}Y\texttt{>} <: N)^?\end{array}}{\Delta \vdash R \in (X :: (\overline{P}))}$$

We show the termination of the judgment under the assumption that R and $\overline{P}$ are well-kinded in the well-kinded environment Δ. In the inlined rule, there is a

subtyping premise $\Delta, \Pi_0 \vdash R@i\texttt{<}Y\texttt{>} <: N$ and a satisfaction premise $\Delta, \Pi \vdash Y \in \Pi'$.

For proving that the subtyping premise is decidable we show that the types $R@i\texttt{<}Y\texttt{>}$ and N are well-kinded in the context Δ, which lets us apply the decidability of subtyping (Lemma 11).

The satisfaction premise $\Delta, \Pi \vdash Y \in \Pi'$ is not problematic since it involves a row parameter Π' which has a smaller kind than the one $((X :: (\overline{P})))$ used in the conclusion of the rule, i.e., we can prove that $|\text{kind}(\Pi')| < |\text{kind}(X :: (\overline{P}))|$.

Lemma 13 [Decidability of type well-formedness]

We show that a type well-formedness judgment like $\Delta \vdash T ::: k$ is decidable provided the environment Δ is well-kinded. The definition of type well-formedness (Fig. 9) recursively depends on itself and on the satisfaction judgment (in rule WF-Apply). Since, for every rule of type well-formedness, the type constructs that appear in the premises are smaller, with respect to their structural size, than the types that appear in the conclusion, the termination of the judgment simply relies on the fact that, in the rule WF-Apply, the judgment $\Delta \vdash R \in \Pi$ is decidable. It has previously been shown that it is actually the case if Δ, R and Π are well-kinded. By hypothesis, Δ is well-kinded. From the premises of the rule WF-Apply, we know that K and R are well-formed. It is easy to prove, by simple induction, that every well-formed type construct is also well-kinded. As a consequence, K and R, which are well-formed, are also well-kinded in the environment Δ. Since K is well-kinded in Δ, and Δ is a well-kinded environment, the parameter Π of K, computed in the context of Δ, is necessarily well-kinded too. This concludes the proof that type well-formedness is decidable.

Lemma 14 [Decidability of typing]

The type assignment judgment (Fig. 10) depends on subtyping, satisfaction, and type well-formedness. These judgments have been shown decidable. Since type assignment is defined in a compositional way, it is also decidable.

Lemma 15 [Decidability of member and class well-formedness]

Well-formedness of members (and consequently of classes and programs) is decidable since it depends on all the other judgments, which have been shown decidable.

D A MORE GENERAL TYPE SYSTEM: FGJ$_\Omega$

For proving that FGJ$_\omega$'s type system is safe, we consider another calculus with a more general type system, called FGJ$_\Omega$. Both calculi have the same abstract syntax

and the same semantics. The differences between them reside in the typing rules. In what follows we just describe the differences. The well-kindedness of types (Fig. 5) and the reduction of types (Fig. 6) stay unchanged. The differences start from the subtyping relation. In order to distinguish FGJ$_\Omega$ judgments from FGJ$_\omega$ ones, we mark them with the subscript Ω.

Differences between FGJ$_\Omega$ and FGJ$_\omega$

- Subtyping in FGJ$_\Omega$ contains an additional rule S-TRANS that states the transitivity of the relation provided the intermediate type is well-kinded.

$$\text{(S-TRANS)}\ \frac{\Delta \vdash S :: * \qquad \Delta \vdash_\Omega T <: S \qquad \Delta \vdash_\Omega S <: U}{\Delta \vdash_\Omega T <: U}$$

- Every sequence of judgments $\text{bound}_\Delta(T) = N$, $\text{superclass}_C(N) = N'$ is replaced with a subtyping judgment $\Delta \vdash T <: N'$ and a well-kindedness judgment $\Delta \vdash N' :: *$. For instance in rules T-SELECT, T-CALL and WF-FIELD-IMPL.

- Generally, a well-formedness judgment is replaced with a well-kindedness judgment. For instance, $\Pi \vdash T ::: *$ is replaced with $\Pi \vdash T :: *$ in rule WF-FIELD-DECL. However, there are two exceptions, which are explained in the following two points.

- Rule T-NEW of FGJ$_\omega$ imposes the condition $\Delta \vdash C\texttt{<}R\texttt{>} ::: *$ in the proof that a term like `new` $C\texttt{<}R\texttt{>}(\bar{t})$ is well-typed. In FGJ$_\Omega$, this premise is replaced with the two premises $\Delta \vdash C\texttt{<}R\texttt{>} :: *$ and $\Delta \vdash_\Omega R \in \Pi$, where Π is the parameter declared by the class C.

- Where it is checked that the superclass $C'\texttt{<}R\texttt{>}$ of a class C is well-formed in rule WF-CLASS-DECL, the premise $\Pi \vdash C'\texttt{<}R\texttt{>} ::: *$ should be replaced with the two premises $\Pi \vdash C'\texttt{<}R\texttt{>} :: *$ and $\Pi \vdash_\Omega R \in \Pi'$, where Π' is the parameter declared by the class C'.

- In FGJ$_\omega$, the rule SAT, which checks the conformance of a type constructor K w.r.t. a parameter P, contains the premise $\Delta, \Pi \vdash X \in \Pi'$ where Π and Π' are respectively the parameters of P and K. In FGJ$_\Omega$, this premise is replaced with $\text{kind}(\Pi) = \text{kind}(\Pi')$.

Lemma 16 [Inclusion of FGJ$_\omega$ inside FGJ$_\Omega$]

In order to derive the type safety of FGJ$_\omega$ from the type safety of FGJ$_\Omega$, we have to prove that every well-formed FGJ$_\omega$ program is a well-formed FGJ$_\Omega$ program. This fact is a corollary of the following implications, which are easily proven by induction on the involved judgments.

• $\Delta \vdash T <: U$	implies	$\Delta \vdash_\Omega T <: U$
• $\Delta \vdash R \in \Pi$	implies	$\Delta \vdash_\Omega R \in \Pi$
• $\Delta \vdash K \in P$	implies	$\Delta \vdash_\Omega K \in P$
• $\Delta \vdash K, R, T ::: k$	implies	$\Delta \vdash K, R, T :: k$
• $\Delta \vdash P, \Pi$ WF	implies	$\Delta \vdash P, \Pi$ WK
• $\Delta; \Gamma \vdash t : T$	implies	$\Delta; \Gamma \vdash_\Omega t : T$
• $C\langle\Pi\rangle \vdash F, M$ WF	implies	$C\langle\Pi\rangle \vdash_\Omega F, M$ WF
• override$(m, C\langle\Pi\rangle, C')$	implies	override$_\Omega(m, C\langle\Pi\rangle, C')$
• $\vdash D$ WF	implies	$\vdash_\Omega D$ WF
• $\vdash p$ WF	implies	$\vdash_\Omega p$ WF

E TYPE SAFETY OF FGJ$_\Omega$ (AND FGJ$_\omega$)

Classically, the proof that FGJ$_\Omega$'s type system is safe is split into a *progress* theorem (well-typed terms that are not values are reducible) and a *subject-reduction* theorem (the type of a term is preserved by reduction).

Admissible rules in FGJ$_\Omega$

Before proving progress and subject reduction we start by showing that FGJ$_\Omega$ satisfies some good properties.

Lemma 17 [Reduction is preserved by type substitution]

$$\frac{T \to T'}{T[X\backslash R] \to T'[X\backslash R]} \qquad \frac{R \to R'}{T[X\backslash R] \xrightarrow{*} T[X\backslash R']}$$

Lemma 18 [Subtyping and satisfaction are preserved by type reduction]

The following rules are admissible.

$$\frac{\Delta \vdash_\Omega T <: U \qquad \Delta \xrightarrow{*} \Delta' \qquad T \xrightarrow{*} T' \qquad U \xrightarrow{*} U'}{\Delta' \vdash_\Omega T' <: U'}$$

$$\frac{\Delta \vdash_\Omega R \in \Pi \qquad \Delta \xrightarrow{*} \Delta' \qquad R \xrightarrow{*} R' \qquad \Pi \xrightarrow{*} \Pi'}{\Delta' \vdash_\Omega R' \in \Pi'}$$

$$\frac{\Delta \vdash_\Omega K \in P \qquad \Delta \xrightarrow{*} \Delta' \qquad K \xrightarrow{*} K' \qquad P \xrightarrow{*} P'}{\Delta' \vdash_\Omega K' \in P'}$$

Proof: Easy. By induction on the derivations of the different judgments using preservation of reduction by type substitution (Lemma 17). **Qed**

Lemma 19 [The subtyping transitivity rule can be eliminated in the empty context]

$\emptyset \vdash_\Omega T <: U$ implies there exists a derivation of the same judgment that does not use the transitivity rule S-TRANS.

Proof: We consider a system of subtyping rules (written $\Delta \vdash_{\Omega\_} T <: U$) that is equivalent to the one of FGJ$_\Omega$ except that it does not contain the transitivity rule S-TRANS. We prove that this rule (reminded below) is actually admissible in this system, by induction on the sequence (T, S, U). The well-founded relation that is used by the induction is $\prec_{\mathsf{seq}}$ (See Lemma 10).

$$\frac{\emptyset \vdash S :: * \qquad \emptyset \vdash_{\Omega\_} T <: S \qquad \emptyset \vdash_{\Omega\_} S <: U}{\emptyset \vdash_{\Omega\_} T <: U}$$

We reason by case analysis on the last rules used to derive $\emptyset \vdash_{\Omega\_} T <: S$ and $\emptyset \vdash_{\Omega\_} S <: U$. To illustrate how the proof proceeds, let us consider the case where the former is S-RED$_{\mathsf{R}}$ and the latter S-CLASS. In this case, S is a class type $C\texttt{<}R\texttt{>}$ and we have at disposal the subtyping assumptions $\emptyset \vdash_{\Omega\_} T <: S'$ and $\emptyset \vdash_{\Omega\_} N[X\backslash R] <: U$ where $C\texttt{<}R\texttt{>} \rightarrow S'$, N is the superclass of C and X the name of its type parameter. We want to prove that $\emptyset \vdash_{\Omega\_} T <: U$.

As it is obtained by reduction of the class type $C\texttt{<}R\texttt{>}$, S' is necessarily a class type $C\texttt{<}R'\texttt{>}$ with $R \rightarrow R'$. From $R \rightarrow R'$ we deduce that $N[X\backslash R] \xrightarrow{*} N[X\backslash R']$ (Lemma 17). Here we need a lemma that states that the subtyping relation without transitivity rule is preserved by reduction (the proof is similar as for Lemma 18). We apply this lemma to the judgment $\emptyset \vdash_{\Omega\_} N[X\backslash R] <: U$, which lets us deduce that $\emptyset \vdash_{\Omega\_} N[X\backslash R'] <: U$. This implies $\emptyset \vdash_{\Omega\_} C\texttt{<}R'\texttt{>} <: U$ by definition of subtyping

(rule S-Class). To sum up, we have $\emptyset \vdash_{\Omega\_} T <: C\texttt{<}R'\texttt{>}$ and $\emptyset \vdash_{\Omega\_} C\texttt{<}R'\texttt{>} <: U$. Since $C\texttt{<}R\texttt{>} \rightarrow C\texttt{<}R'\texttt{>}$, we have $\emptyset \vdash C\texttt{<}R\texttt{>} \prec C\texttt{<}R'\texttt{>}$, by definition of type expansion (rule E-Red). By consequence, the sequence $(T, C\texttt{<}R'\texttt{>}, U)$ is smaller, w.r.t. $\prec_{\texttt{seq}}$, than the sequence $(T, C\texttt{<}R\texttt{>}, U)$. It means we can apply the induction hypothesis and conclude that $\emptyset \vdash_{\Omega\_} T <: U$. **Qed**

Progress in FGJ$_\Omega$

Lemma 20 [Subtyping of class types is compatible with subclassing (in the empty context)]

$\emptyset \vdash_\Omega C\texttt{<}R\texttt{>} <: C'\texttt{<}R'\texttt{>}$ implies C is a subclass of C'.

Proof: By Lemma 19, there exists a derivation of $\emptyset \vdash_\Omega C\texttt{<}R\texttt{>} <: C'\texttt{<}R'\texttt{>}$ that does not contain the rule S-Trans, i.e., the judgment $\emptyset \vdash_{\Omega\_} C\texttt{<}R\texttt{>} <: C'\texttt{<}R'\texttt{>}$ is derivable. By induction on this judgment it is easy to show that C is a subclass of C'. **Qed**

Lemma 21 [Progress]

If $\emptyset; \emptyset \vdash_\Omega t : T$ and t is not a value, then there exists u such that $t \rightarrow u$.

Proof: By induction on $\emptyset; \emptyset \vdash_\Omega t : T$, using Lemma 20. Let us consider for example the case T-Select where t is a field selection $t'.f$. If t' is not a value, by (IH) there exists u' such $t' \rightarrow u'$ and by rule R-Context we deduce that $t'.f \rightarrow u'.f$. If t' is a value $\texttt{new}\ C\texttt{<}R\texttt{>}(\overline{v})$, since the term $t'.f$ is well-typed, we have $\mathrm{fields}(C) = \overline{f}$ with $|\overline{f}| = |\overline{v}|$, and $\emptyset \vdash_\Omega C\texttt{<}R\texttt{>} <: C'\texttt{<}R'\texttt{>}$ with $(U\ f;) \in C'$. By Lemma 20, $\emptyset \vdash_\Omega C\texttt{<}R\texttt{>} <: C'\texttt{<}R'\texttt{>}$ implies C is a subclass of C'. C is a subclass of C' implies $\mathrm{fields}(C') \subset \mathrm{fields}(C) = \overline{f}$. Since $(U\ f;) \in C'$ and $\mathrm{fields}(C') \subset \overline{f}$, necessarily $f \in \overline{f}$, i.e. there exists i such that $f = f_i$. By rule R-Select, we conclude that $\texttt{new}\ C\texttt{<}R\texttt{>}(\overline{v}).f$ reduces to v_i. **Qed**

Subject reduction in FGJ$_\Omega$

Lemma 22 [Subtyping is preserved by type substitution]

Let $\Pi = X :: (\overline{P})$. Assume $\Delta \vdash R :: k$, $\Delta \vdash_\Omega R \in \Pi$ and $\vdash \Delta, \Pi, \Delta'$ wk.

$$\frac{\Delta, \Pi, \Delta' \vdash T, U :: * \qquad \Delta, \Pi, \Delta' \vdash_\Omega T <: U}{\Delta, \Delta'[X \backslash R] \vdash_\Omega T[X \backslash R] <: U[X \backslash R]}$$

Proof: The proof is by induction on the size of Π's kind ($|\text{kind}(\Pi)|$, followed by an induction on the derivation depth of the subtyping judgment $\Delta, \Pi, \Delta' \vdash_\Omega T <: U$. We reason by case analysis on the last rule that was used to derive this judgment. For conciseness, we just describe the case that needs the induction hypothesis about the size of kind(Π). It is a subcase of the rule S-VAR where the variable composing T is exactly the one that R is substituted for, i.e. $T = X@i\texttt{<}R'\texttt{>}$. We have to prove that $\Delta, \Delta'[X\backslash R] \vdash_\Omega R@i\texttt{<}R'[X\backslash R]\texttt{>} <: U[X\backslash R]$. The hypotheses coming from rule S-VAR are $P_i = \texttt{<}\Pi'\texttt{>} \triangleleft N$, $\Pi' = Y :: (\overline{Q})$ and $\Delta, \Pi, \Delta' \vdash_\Omega N[Y\backslash R'] <: U$.

(1) We substitute R for X in the last judgment by applying the hypothesis generated by the second induction. The IH applies because the judgment has a smaller derivation depth. We obtain $\Delta, \Delta'[X\backslash R] \vdash_\Omega N[Y\backslash R'][X\backslash R] <: U[X\backslash R]$.

(2) By weakening the hypothesis $\Delta \vdash_\Omega R \in \Pi$ we obtain $\Delta, \Delta'[X\backslash R] \vdash_\Omega R \in \Pi$. By inversing the rules POLY-SAT and SAT, used to derive this satisfaction judgment, and from the definitions of Π, P_i and Π' we deduce that $\Delta, \Delta'[X\backslash R], \Pi_0 \vdash_\Omega R@i\texttt{<}Y\texttt{>} <: N[X\backslash R]$ where $\Pi_0 = \text{erase}(\Pi'[X\backslash R])$.

(3) By inversion of the hypothesis $\Delta, \Pi, \Delta' \vdash X@i\texttt{<}R'\texttt{>} :: *$ we deduce that $\Delta, \Pi, \Delta' \vdash R' :: \text{kind}(\Pi')$. Using Lemma 1 we substitute R for X in this judgment, which gives $\Delta, \Delta'[X\backslash R] \vdash R'[X\backslash R] :: \text{kind}(\Pi')$.

(4) It follows from the definition of Π_0 that $\text{kind}(\Pi_0) = \text{kind}(\Pi'[X\backslash R]) = \text{kind}(\Pi')$, so (3) becomes $\Delta, \Delta'[X\backslash R] \vdash R'[X\backslash R] :: \text{kind}(\Pi_0)$. Since there is no bound to satisfy in Π_0, this last judgment implies that $\Delta, \Delta'[X\backslash R] \vdash_\Omega R'[X\backslash R] \in \Pi_0$.

(5) We substitute $R'[X\backslash R]$ for Y in (2) by applying the hypothesis generated by the first induction. The IH applies because $|\text{kind}(\Pi_0)| < |\text{kind}(\Pi)|$. We obtain $\Delta, \Delta'[X\backslash R] \vdash_\Omega R@i\texttt{<}R'[X\backslash R]\texttt{>} <: N[X\backslash R][Y\backslash R'[X\backslash R]]$.

(6) We remark that $N[Y\backslash R'][X\backslash R] = N[X\backslash R][Y\backslash R'[X\backslash R]]$ and by applying transitivity of subtyping (rule S-TRANS) to (1) and (5) we conclude the proof. **Qed**

Lemma 23 [Satisfaction is preserved by type substitution]

Let $\Pi = X :: (\overline{P})$. Assume $\Delta \vdash R :: k$, $\Delta \vdash_\Omega R \in \Pi$ and $\vdash \Delta, \Pi, \Delta'$ WK.

$$\frac{\Delta, \Pi, \Delta' \vdash K :: k' \qquad \Delta, \Pi, \Delta' \vdash P \text{ WK} \qquad \Delta, \Pi, \Delta' \vdash_\Omega K \in P}{\Delta, \Delta'[X\backslash R] \vdash_\Omega K[X\backslash R] \in P[X\backslash R]}$$

$$\frac{\Delta, \Pi, \Delta' \vdash R' :: k' \qquad \Delta, \Pi, \Delta' \vdash \Pi' \text{ WK} \qquad \Delta, \Pi, \Delta' \vdash_\Omega R' \in \Pi'}{\Delta, \Delta'[X\backslash R] \vdash_\Omega R'[X\backslash R] \in \Pi'[X\backslash R]}$$

Proof: The first property is a corollary of Lemma 22. The second property is a corollary of the first one. **Qed**

Lemma 24 [Typing is preserved by type substitution]

Let $\Pi = X :: (\overline{P})$. Assume $\Delta \vdash R :: k$, $\Delta \vdash_\Omega R \in \Pi$ and $\vdash \Delta, \Pi, \Delta'$ WK.

$$\frac{\Delta, \Pi, \Delta'; \Gamma \vdash_\Omega t : T}{\Delta, \Delta'[X\backslash R]; \Gamma[X\backslash R] \vdash_\Omega t[X\backslash R] : T[X\backslash R]}$$

Proof: By induction on the type assignment judgment using Lemma 1, Lemma 22 and Lemma 23. **Qed**

Lemma 25 [Typing is preserved by term substitution]

$$\frac{\emptyset; x{:}U, \Gamma \vdash_\Omega t : T \qquad \emptyset; \emptyset \vdash_\Omega v : S \qquad \emptyset \vdash_\Omega S <: U}{\exists T', \quad \emptyset; \Gamma \vdash_\Omega t[x\backslash v] : T' \ \wedge \ \emptyset \vdash_\Omega T' <: T}$$

Proof: By induction on the type assignment judgment. **Qed**

Lemma 26 [Typing is preserved by environment reduction]

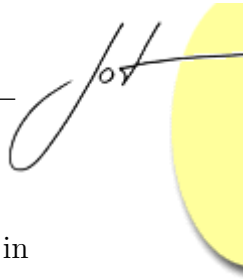
$$\frac{\Delta; \Gamma \vdash_\Omega t : T \qquad \Delta \xrightarrow{*} \Delta'}{\Delta'; \Gamma \vdash_\Omega t : T} \qquad \frac{\Delta; \Gamma \vdash_\Omega t : T \qquad \Gamma \xrightarrow{*} \Gamma'}{\exists T', \quad \Delta; \Gamma' \vdash_\Omega t : T' \ \wedge \ \Delta \vdash_\Omega T' <: T}$$

Proof: The first property is proven by induction on type assignment using Lemma 18. The second property is proven by induction on type assignment using transitivity of subtyping (rule S-TRANS) and Lemma 18. **Qed**

Lemma 27 [Typing is preserved by term reduction (= subject-reduction)]

$$\frac{\emptyset; \emptyset \vdash_\Omega t : T \qquad t \rightarrow t'}{\exists T', \quad \emptyset; \emptyset \vdash_\Omega t' : T' \ \wedge \ \emptyset \vdash_\Omega T' <: T}$$

Proof: Easy but tedious. By induction on the judgment $\emptyset; \emptyset \vdash_\Omega t : T$ using all the lemmas proven in this section. The lemma can be generalized to an arbitrary number of reduction steps from t to t' (by induction on the number of steps using rule S-TRANS). **Qed**



Lemma 28 [Type safety of FGJ$_\Omega$]

If p is a well-formed FGJ$_\Omega$ program, t is its main expression, and t reduces in zero or more steps to an irreducible term u, then u is a value.

Proof: Immediate using Lemma 21 (progress) and Lemma 27 (subject-reduction). **Qed**

Lemma 29 [Type safety of FGJ$_\omega$]

If p is a well-formed FGJ$_\omega$ program, t is its main expression, and t reduces in zero or more steps to an irreducible term u, then u is a value.

Proof: By Lemma 16, every well-formed FGJ$_\omega$ program is a well-formed FGJ$_\Omega$ program. We conclude using Lemma 28. **Qed**