Adding Type Constructor Parameterization to Java

Philippe Altherr and Vincent Cremet

Abstract. 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 and motivate our extension through two examples: the definition of generic data-types with binary methods and the definition of generalized algebraic data-types. The formalization and the safety and decidability proofs are, for space reasons, only shortly described.

Introduction

Most mainstream programming languages let the programmer parameterize his or her data structures and algorithms by *types*. The general term for this mechanism is parametric polymorphism[1]. It allows the same piece of code to be used with different type instantiations. Some languages, like HASKELL [1], additionally let the programmer parameterize his or her 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 (a data-type which is itself parameterized by a type). Although this mechanism has been widely recognized as useful, for example to represent monads [2] as a library, no attempt has been made until now, to our knowledge, to design and implement a similar feature for JAVA.

A *type constructor* is a function that takes a list of types (in full generality, a list of type constructors) and returns a type. For instance, the JAVA class `List` of the standard library is a type constructor; it can be applied to a type T with the syntax $\texttt{List}\langle T\rangle$ 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. Binary methods [3] are a well-known challenge for object-oriented programming. Section 1 first recalls the problem posed by binary methods. Then, it describes a technique based on F-bounded polymorphism that can sometimes be used to solve it. Finally,

[1] In some contexts it is called differently: for instance, generics in JAVA or templates in C++.

it shows how type constructor parameterization can be used to generalize this technique to generic data-types. Section 2 describes how the Visitor design-pattern can be enhanced through the use of type constructor parameterization to implement generalized algebraic data-types. Section 3 describes our FGJ$_\omega$ calculus, which formalizes type constructor parameterization and our prototype compiler, which implements it. Section 4 discusses the safety and decidability of FGJ$_\omega$. Section 5 reviews related work and our contributions and concludes on possible continuations.

1 Generic Data-types with Binary Methods

A *binary method* is a method whose signature contains occurrences of the current class in contravariant positions, for example as an argument type. A problem arises if in a subclass these occurrences need to be replaced with occurrences of the subclass in order to specialize the method. Indeed, it is well-known that it is unsafe and thus forbidden to covariantly refine types that occur in contravariant positions.

The problem is illustrated below by the binary method `lessThan` of the class `Ordered`. In order to implement it in the subclass `Integer`, it is necessary to have access to the field `i` of the parameter `that`. However, changing the type of `that` to `Integer` amounts to a covariant change of a type in a contravariant position, which is unsafe and thus forbidden.

```
abstract class Ordered {
  abstract boolean lessThan(Ordered that);
}
class Integer extends Ordered { int i;
  boolean lessThan(Integer that) {          // illegal
    return this.i ≤ that.i; }
}
```

A classical and simple technique for solving this problem is to use F-bounded polymorphism [4]; the base class is parameterized by a type `Self` and occurrences of the base class in the signature of the binary method are replaced with `Self`. The type `Self` represents the exact type of the current object, also called its *self type*. It cannot be known exactly since the base class is open for subclassing. It is only known that the self type is at least a subtype of the base class. In our example, this is expressed by the bound `Ordered⟨Self⟩`. The fact that `Self` appears in its own bound is the essence of F-bounded polymorphism.

```
abstract class Ordered⟨Self extends Ordered⟨Self⟩⟩ {
  abstract boolean lessThan(Self that);
}
class Integer extends Ordered⟨Integer⟩ { int i;
  boolean lessThan(Integer that) { return this.i ≤ that.i; }
}
```

The subclass `Integer` instantiates the type parameter `Self` to itself. This leads to a signature for the method `lessThan` that is compatible with its implementation.

The above technique does not always directly apply when the base class is generic. To see why, we consider a class `Collection`, representing immutable collections of objects, that is parameterized by the type `X` of its elements. For our purpose, a collection declares just two methods: `append` and `flatMap`. The first one merges into a new collection the receiving one and the one passed as an argument. The second method applies the functions passed as an argument to each element of the receiving collection and merges all returned collections into a new one.

```
abstract class Function⟨X,Y⟩ { abstract Y apply(X x); }
abstract class Collection⟨X⟩ {
  abstract     Collection⟨X⟩ append(Collection⟨X⟩ that);
  abstract ⟨Y⟩ Collection⟨Y⟩ flatMap(Function⟨X,Collection⟨Y⟩⟩ f);
}
```

Both methods are binary methods because the current class occurs in the type of their parameter. Both methods also raise similar problems as the method `lessThan` when they are implemented in subclasses. This is illustrated below by the subclass `List`, which implements linked lists.

```
class List⟨X⟩ extends Collection⟨X⟩ {
          List()               { ... } // construct an empty list
  boolean isEmpty()            { ... } // test emptiness
  X       head()               { ... } // get first element
  List⟨X⟩ tail()               { ... } // get all elements but the first
  List⟨X⟩ add(X that)          { ... } // add an element
  List⟨X⟩ append(List⟨X⟩ that) {                   // illegal
    return isEmpty() ? that : tail().append(that).add(head());
  }
  ⟨Y⟩ List⟨Y⟩ flatMap(Function⟨X,List⟨Y⟩⟩ f) {     // illegal
    return isEmpty() ? new List⟨Y⟩()
                     : f.apply(head()).append(tail().⟨Y⟩flatMap(f));
  }
}
```

As for the method `lessThan`, we have been forced to change the signature of both binary methods by replacing each occurrence of `Collection` with `List`. Indeed, if the method `append` did not return an instance of `List`, the call to `add` would be illegal and if its parameter was not an instance of `List`, returning `that` would be illegal. Similarly, if the method `flatMap` did not return an instance of `List`, passing `tail().⟨Y⟩flatMap(f)` as argument to `append` would be illegal and if the function `f` did not return instances of `List`, calling the method `append` on `f.apply(head())` would be illegal[2]. The covariant change of the return types is safe, but the covariant change of the argument types is not.

If we try to encode the self type of the class `Collection` with the same technique as for the class `Ordered`, we obtain the following definition.

```
abstract class Collection⟨Self extends Collection⟨Self,X⟩,X⟩ {
  abstract      Collection⟨Self,X⟩ append(Self that);
  abstract ⟨Y⟩ Collection⟨Self,Y⟩ flatMap(Function⟨X,???⟩ f);
}
```

This solves the problem for the method `append` (provided the superclass of `List` is replaced with `Collection⟨List⟨X⟩,X⟩`) but this is helpless to express the type of the parameter `f` of `flatMap`. And worse, even the return type of `flatMap` seems dubious now as it describes a collection of `Y`s whose self type is a collection of `X`s. The problem of `flatMap` is that it instantiates the class type `Collection` with a type `Y` which is a priori different from the parameter `X` of the class. Here, we need more than a self type, we need a self type *constructor*; `Self` should represent a type constructor instead of a type. This is expressed in the following definition.

```
abstract class Collection⟨Self⟨Z⟩ extends Collection⟨Self,Z⟩,X⟩ {
  abstract      Collection⟨Self,X⟩ append(Self⟨X⟩ that);
  abstract ⟨Y⟩ Collection⟨Self,Y⟩ flatMap(Function⟨X,Self⟨Y⟩⟩ f);
}
```

In this definition, the class `Collection` declares that its first parameter is a unary type constructor named `Self` such that when it is applied to some type `Z` it returns a subtype of `Collection⟨Self,Z⟩`. Here, the type of the argument of both methods can be correctly expressed and their implementation in the class `List` is now possible provided its superclass is replaced with `Collection⟨List,X⟩`.

```
class List⟨X⟩ extends Collection⟨List,X⟩ { ...
        List⟨X⟩ append(List⟨X⟩ that) { ... }
  ⟨Y⟩ List⟨Y⟩ flatMap(Function⟨X,List⟨Y⟩⟩ f) { ... }
}
```

2 Generalized Algebraic Data-types

An *algebraic* data-type is a type that is inductively defined by the union of different cases; functions operating on objects of an algebraic data-type can be defined by pattern-matching on the different cases. A generic algebraic data-type is called a *generalized algebraic data-type* (GADT) [5] when different cases in its

[2] Strictly speaking, this is true only if the method `append` is not declared in the base class `Collection` but only in the class `List`.

definition instantiate its type parameter with different types. Indeed, such an algebraic data-type needs a generalization of the way pattern-matching constructs are typed in order to fully exploit its type parameter. We present a solution to this problem that is based on the Visitor design-pattern [6], enhanced with type constructor parameterization.

The classical example to illustrate GADTs is a safe evaluator for a simple programming language. In this example, an algebraic data-type `Expr` is used to represent the expressions of the language. The different cases of the algebraic data-type correspond to the different constructs of the language (constants, arithmetic operations, conditionals, etc). The type `Expr` is parameterized by a type `X`; for a given expression, `X` represents the type of the value returned by its evaluation. Thus, the case `IntLit`, for literal integers, instantiates `X` to `Integer`, while the case `If`, for conditionals, instantiates `X` to the same type as its two branches (then & else). A first benefit of encoding the type of the evaluation of an expression in the type of its representation is to enable the type-checker to reject ill-formed expressions like `new Plus(new IntLit(2), new BoolLit(false))`.

Following the structure of the visitor design-pattern, we start by defining a base class for expressions and one for visitors.

```
abstract class Expr⟨X⟩ {
  abstract ⟨R⟨_⟩⟩ R⟨X⟩ accept(Visitor⟨R⟩ v);
}
abstract class Visitor⟨R⟨_⟩⟩ {
  abstract R⟨Integer⟩ caseIntLit(int x);
  abstract R⟨Boolean⟩ caseBoolLit(boolean x);
  abstract R⟨Integer⟩ casePlus(Expr⟨Integer⟩ x, Expr⟨Integer⟩ y);
  abstract R⟨Boolean⟩ caseCompare(Expr⟨Integer⟩ x, Expr⟨Integer⟩ y);
  abstract ⟨X⟩ R⟨X⟩ caseIf(Expr⟨Boolean⟩ x, Expr⟨X⟩ y, Expr⟨X⟩ z);
}
```

The method `accept`, which applies a visitor to an expression of type `Expr⟨X⟩`, is parameterized by a type constructor `R` and returns an object of type `R⟨X⟩`. It means that the type returned by the matching of an expression will functionally depend on the type `X` that characterizes the expression. The syntax `R⟨_⟩` indicates that `R` stands for a unary type constructor and that we are not interested in naming its argument. The class `Visitor` is standard except for the fact that it is parameterized by a type constructor, which is used in the return type of its methods.

Finally, there is one subclass of `Expr` for each kind of expression. In these classes the implementation of the method `accept` simply forwards the class attributes to the corresponding method of the visitor argument. For conciseness, we give only two representative subclasses.

```
class IntLit extends Expr⟨Integer⟩ { int x;
  ⟨R⟨_⟩⟩ R⟨Integer⟩ accept(Visitor⟨R⟩ v) { return v.caseIntLit(x); }
}
class If⟨Y⟩ extends Expr⟨Y⟩ { Expr⟨Boolean⟩ x; Expr⟨Y⟩ y; Expr⟨Y⟩ z;
  ⟨R⟨_⟩⟩ R⟨Y⟩ accept(Visitor⟨R⟩ v) { return v.⟨Y⟩caseIf(x, y, z); }
}
```

A safe evaluator for our language of expressions will maintain the consistency between the type of an expression and the type of the values to which it evaluates. It is implemented as a polymorphic method `eval` that takes an expression `e` of type `Expr⟨T⟩` and returns a value of type `T`, as shown below.

```
class Eval extends Visitor⟨⟨Y⟩ ⇒ Y⟩ {
  ⟨T⟩ T eval(Expr⟨T⟩ e) { return e.⟨⟨Y⟩ ⇒ Y⟩accept(this); }

  Integer caseIntLit(int x) { return new Integer(x); }
   ...
  ⟨X⟩ X caseIf(Expr⟨Boolean⟩ x, Expr⟨X⟩ y, Expr⟨X⟩ z) {
    return this.⟨Boolean⟩eval(x).booleanValue()
      ? this.⟨X⟩eval(y);
      : this.⟨X⟩eval(z);
    }
}
```

The method `eval` is implemented as a member of a `Visitor` subclass whose parameter `R` is instantiated with the type constructor `⟨Y⟩ ⇒ Y`. Such an expression is called an *anonymous type constructor*. It represents an anonymous function over types. In our example, the anonymous type constructor `⟨Y⟩ ⇒ Y` is the identity function, it takes a type `Y` as argument and returns the same type `Y`.

Let us check that the members of the class `Eval` are well-typed. In the method `eval`, since `e` is of type `Expr⟨T⟩` the type returned by the call to the method `accept` is `R⟨T⟩` with `R` instantiated to `⟨Y⟩ ⇒ Y`, i.e., `(⟨Y⟩ ⇒ Y)⟨T⟩`. This type reduces in one step to `T`, which is indeed the return type declared by the method `eval`. In the class `Visitor` the method `caseIntLit` is defined with the return type `R⟨Integer⟩`. In the class `Eval`, `R` is instantiated to `⟨Y⟩ ⇒ Y`, thus `caseIntLit` should return a value of type `(⟨Y⟩ ⇒ Y)⟨Integer⟩` (i.e., `Integer`), which is indeed the case. Similarly one can check that the other methods of the visitor return values of the right type.

3 Formalization: The FGJ$_\omega$ Calculus

The calculus FGJ$_\omega$ (pronounce "FGJ-omega") is a formalization of our design for type constructor parameterization. It is an extension of Featherweight Generic Java [7] (FGJ), which is a core language for JAVA with a focus on generics. Our calculus enhances FGJ by replacing all parameters representing types by parameters representing type constructors. Its name is derived from an analogy with λ_ω [8], which enhances the simply typed lambda-calculus [9] with type operators.

The syntax of FGJ$_\omega$ and FGJ are very similar. The main change is that parameters and arguments representing types are replaced with parameters and arguments representing type constructors. The elements of the syntax that differ are summarized below.

type parameter	$P ::= X\langle\overline{P}\rangle\ \texttt{extends}\ N$
type constructor	$K ::= X \mid C \mid \langle\overline{P}\rangle \Rightarrow T$
type	$T ::= K\langle\overline{K}\rangle$
class type	$N ::= C\langle\overline{K}\rangle$

A type parameter declaration P always represents a type constructor. Its syntax $X\langle\overline{P}\rangle\ \texttt{extends}\ N$ specifies that the parameter is named X, that the type constructor it represents accepts arguments that conform to the parameters $\overline{P}$ and that the type it returns when it is applied to such arguments conforms to the type N. A type constructor K is either a type parameter X, a class C, or an anonymous type constructor $\langle\overline{P}\rangle \Rightarrow T$ that expects arguments conforming to the parameters $\overline{P}$ and that returns the type T. This last construct is analogous to anonymous functions in functional programming languages except that it operates at the level of types and not at the level of values. All types T consist of a type constructor K applied to a list of type constructors $\overline{K}$. Class types N are a restriction of types T used as upper-bounds of type parameters in order to guarantee the decidability of subtyping. All other syntactic constructs of FGJ$_\omega$ are identical to those of FGJ except that all sequences $\overline{X} \triangleleft \overline{T}$ of type parameters (in class and method declarations) are replaced with sequences of parameters $\overline{P}$.

Modifying FGJ's typing rules to take into account parameters representing type constructors is relatively straightforward. The main issue is that, because of anonymous type constructors, the subtyping judgement must now include the reduction of types. For instance, `(⟨X⟩ ⇒ List⟨X⟩)⟨Integer⟩` must be considered a subtype of `List⟨Integer⟩` since the former reduces in one replacement step to the latter.

In our examples, we have types that consist of a simple class like `Integer` or a simple parameter like `X` and other that consists of a type constructor applied to a list of arguments; but our calculus supports only the latter ones. Similarly, in our examples, we have both parameters that represent types and parameters that represent type constructors; but again FGJ$_\omega$ supports only the latter ones. This apparent contradiction is not one if one considers that parameters representing types represent in fact type constructors with zero arguments whose empty lists of parameters have been omitted and that types without type arguments are in fact applications of type constructors with zero arguments whose empty lists of arguments have been omitted. Thus, `Integer` and `X` are syntactic sugar for `Integer⟨⟩` and `X⟨⟩` and a type parameter declaration like `Self⟨Z⟩ extends Collection⟨Self,Z⟩` is syntactic sugar for

```
Self⟨Z⟨⟩ extends Object⟨⟩⟩ extends Collection⟨Self,Z⟩
```

This declaration defines a unary type constructor parameter named `Self`. It specifies that the domain of `Self`, i.e., the set of arguments for which it is well-defined, contains the set $\mathbb{S}$ of type constructors Z_1 taking no arguments such that $Z_1\langle\rangle$ is a subtype of $\texttt{Object}\langle\rangle$. The declaration also specifies that the type constructor `Self` must satisfy the property that $\texttt{Self}\langle Z_2\rangle$ is a subtype of $\texttt{Collection}\langle\texttt{Self,}Z_2\rangle$ if Z_2 is an element of $\mathbb{S}$.

As exemplified by this declaration, the domain of a type constructor parameter (`Self`) is syntactically expressed by other type constructor parameters ($\texttt{Z}\langle\rangle$ $\texttt{extends Object}\langle\rangle$). This recursiveness in the syntax of type constructor parameters is one reason why their declarations are so compact. The other reason is the versatility of binders: in the above declaration, the name `Z` is a binder that serves simultaneously two purposes, (1) defining the domain of `Self` and (2) defining the upper-bound of `Self`. In the above description of the parameter `Self` we made this double role explicit by using a different variable for each role, namely Z_1 and Z_2.

Implementation All examples of this paper have been tested by our prototype FGJ$_\omega$ compiler [10] (actually, just a type-checker and an interpreter for now). It integrates all the syntactic sugar described above. In addition to that, it replaces types that are in position of type constructors with parameterless anonymous functions, so that a type like $\texttt{List}\langle\texttt{List}\langle\texttt{Integer}\rangle\rangle$ is internally represented by $\texttt{List}\langle\langle\rangle \Rightarrow \texttt{List}\langle\texttt{Integer}\rangle\rangle$.

4 Theoretical Study of FGJ$_\omega$

The type system of FGJ$_\omega$ is both safe and decidable. The proofs of these properties can be found in a companion paper [11]. Here we outline the main arguments of the proofs.

Safety By definition, FGJ$_\omega$'s type system is *safe* if every well-formed FGJ$_\omega$ program is safe. A FGJ$_\omega$ program is safe if every time a member (method or field) is selected on an object at runtime, it will be found in the object.

Our proof of type safety is based on the small-step operational semantics that comes with FGJ. Classically [12], the proof is split into a progress theorem and a subject-reduction (type preservation) theorem. These properties are proven on a more general type system, called FGJ$_\Omega$, with the same syntax and semantics as FGJ$_\omega$ but different (less constraining) typing rules. Compared to the original type system (FGJ$_\omega$), FGJ$_\Omega$ alleviates some constraints that were introduced to ensure decidability. More importantly, while checking that a type application $K\langle\overline{K}\rangle$ is well-formed, FGJ$_\Omega$ never tests that the arguments $\overline{K}$ satisfy the subtyping conditions expected by the parameters of K, it only tests that the arities are appropriate. It turns out that the test of subtyping conformance has no impact on type safety and that its sole role is to disallow the definition of empty types. By removing this test, type well-formedness no longer depends on subtyping and the proof becomes easier. Finally, the fact that every well-formed

FGJ$_\omega$ program is also a well-formed FGJ$_\Omega$ program allows us to derive the type safety of FGJ$_\omega$ from the type safety of FGJ$_\Omega$.

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

The more challenging judgement 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 judgement. So, what prevents the type-checker from being caught in an infinite reduction sequence? Actually, it can be shown 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 [13, 14]. This is not surprising since well-kinded FGJ$_\omega$ types are almost isomorphic to well-typed lambda-terms. The only difference is the presence of type bounds attached to FGJ$_\omega$'s type parameters and the fact that these bounds are reducible. For example, the anonymous type constructor $\langle X \texttt{ extends } T\rangle \Rightarrow U$ reduces to $\langle X \texttt{ extends } T'\rangle \Rightarrow U$ if T reduces to T'. There is no similar reduction step in the lambda-calculus.

However, this difference between both formalisms does not prevent us from interpreting FGJ$_\omega$ types as lambda-terms. The solution is to consider an extension of the simply typed lambda-calculus with tuples, for which strong normalization still holds. Then, we interpret well-kinded FGJ$_\omega$ types as well-typed lambda-terms thanks to a translation function $[\![\cdot]\!]$ whose interesting cases are the following ones.

$$\begin{aligned}
[\![\langle\overline{P}\rangle \Rightarrow T]\!] &= \lambda\overline{X} : \mathrm{type}(\overline{P}).([\![T]\!], [\![\overline{P}]\!]) \\
[\![K\langle\overline{K}\rangle]\!] &= \mathbf{fst}\,([\![K]\!]\,[\![\overline{K}]\!])
\end{aligned}$$

In these translations, $\overline{X}$ stands for the variables declared by $\overline{P}$, type($\overline{P}$) denotes a suitable type obtained from $\overline{P}$ and **fst** returns the first component of a tuple.

We can easily prove that each time a FGJ$_\omega$ type T reduces to U, the lambda-term $[\![T]\!]$ reduces to $[\![U]\!]$. Suppose now, by contradiction, there exists an infinite sequence of reductions starting from a well-kinded type T, then there exists an infinite sequence of reductions starting from the well-typed term $[\![T]\!]$, which contradicts the strong normalization of lambda-terms.

5 Conclusion

Related work The combination of subtyping and higher-order polymorphism has been extensively studied in the context of lambda-calculi and object-calculi under the name of *higher-order subtyping* ($F^{\omega}_{<:}$ [15], $F^{\omega}_{\leq}$ [16], $\mathrm{O}\textsc{b}_{\omega<:\mu}$ [17]). The originality of our work is to consider a calculus that is close to a real programming language so that no encoding is needed, neither for objects nor for classes. Furthermore, our calculus incorporates the features of F-bounded polymorphism (the property of type parameter declarations to be mutually recursive in their bounds) and nominal subtyping, which are not covered by any of the calculi cited above. Like these calculi, FGJ_{ω} does not identify eta-convertible type constructors like `List` and $\langle$`X`$\rangle \Rightarrow$ `List`$\langle$`X`$\rangle$. It is left to the programmer the responsibility of writing types in such a manner that the compiler never has to compare such type constructors.

The example presented in Section 1 is nothing but what HASKELL programmers call monads [2], 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 (`Collection`), while it is an external function (written `>>=` and called "bind") in HASKELL.

In Section 2, our encoding of GADTs relies on an extension of JAVA with type constructor parameterization. In [5], the authors found a similar solution by considering an extension with type equality constraints.

SCALA's type system [18] is able to express some use cases of type constructor parameterization [19, 20]. The principle is to encode the declaration of a parameter representing a type constructor like `X`$\langle$`_`$\rangle$ as the declaration of an abstract type (type parameter or virtual type) X bounded by Arity1, where Arity1 is a class with a single type field (say A1). A type like `X`$\langle$`String`$\rangle$ can then be expressed in SCALA by the *refined type* X{**type** A1 = String}, which represents all instances of X in which the type field A1 is equal to String. It is an open question whether FGJ_{ω} can be entirely encoded this way, especially if more complex bounds and anonymous type constructors are involved.

Shortly after we implemented and made public a prototype compiler for FGJ_{ω}, type constructor parameterization was independently integrated to the SCALA compiler under the name of type constructor polymorphism [21]. 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 allows 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_{ω}'s anonymous type constructors since they are missing in SCALA. As for FGJ_{ω}, 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). For the subset of SCALA that corresponds to FGJ_{ω}, our work can be thought of as a theoretical foundation.

Contrary to our calculus, there is no syntax for anonymous type constructors in HASKELL. However a restricted form of anonymous type constructors exist internally; they arise from partial applications of type constructors. For instance, HASKELL lets the programmer write 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 $\langle \mathtt{X} \rangle \Rightarrow \mathtt{Pair}\langle \mathtt{A},\mathtt{X} \rangle$ in our design. Thus, HASKELL is able to represent all anonymous type constructors that correspond to partial type applications but not the others. This is clearly a restriction compared to our design: for example, the FGJ$_\omega$ anonymous type constructor $\langle \mathtt{X} \rangle \Rightarrow \mathtt{Pair}\langle \mathtt{X},\mathtt{A} \rangle$ 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, because higher-order unification is known to be undecidable. Since we adopt in FGJ$_\omega$ the JAVA philosophy of explicit type annotations along with some type inference (in fact none in our prototype), we are not bound to this limitation. In [22], the authors study an extension of HASKELL with functions at the level of types, which shows there is an interest for this feature even in the HASKELL context.

Summary This paper describes a simple way of adding type constructor parameterization to JAVA. The main elements of this extension, a generalized syntax for type parameters and generalized typing rules, have been gradually introduced through two concrete examples that cannot be expressed just with type parameterization. These examples are the definition of generic data-types with binary methods and the definition of generalized algebraic data-types. Our extension, called FGJ$_\omega$, has been formalized, implemented in a prototype compiler, 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. It constitutes also the first proof of safety for a type system that mixes higher-order polymorphism and F-bounded polymorphism.

Future work Our calculus still misses some useful features of JAVA-like languages. Wildcard types [23, 24] have been popularized by their implementation in JAVA. A continuation of this work would be to study, at a theoretical level, the interaction between type constructor parameters and wildcard types. From a more pragmatic point of view, it would also be interesting to see whether type constructor parameters are compatible with the most common strategies for type argument inference in JAVA-like languages.

References

1. Jones, S.P.: The Haskell 98 language and libraries: The revised report. Cambridge University Press (2003)
2. Wadler, P.: Monads for functional programming. In Broy, M., ed.: 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.

3. Bruce, K.B., Cardelli, L., Castagna, G., the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Valery Trifonov), Leavens, G.T., Pierce, B.: On binary methods. Theory and Practice of Object Systems **1**(3) (1996) 221–242
4. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: Proceedings of the fourth international conference on Functional programming languages and computer architecture (FPCA'89), New York, NY, USA, ACM Press (1989) 273–280
5. Kennedy, A., Russo, C.: Generalized algebraic data types and object-oriented programming. In: ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), ACM Press (October 2005)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts (1994)
7. Igarashi, A., Pierce, B., Wadler, P.: 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.
8. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
9. Church, A.: A formulation of the simple theory of types. Journal of Symbolic Logic **5** (1940) 56–68
10. Altherr, P., Cremet, V.: FGJ-omega web page. http://lamp.epfl.ch/∼cremet/FGJ-omega/
11. Cremet, V.: FGJ-omega: Definitions and proofs. http://lamp.epfl.ch/∼cremet/FGJ-omega/FGJ-omega-metatheory.pdf (April 2007)
12. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115** (1994)
13. Tait, W.W.: Intensional interpretations of functionals of finite type I. Journal of Symbolic Logic **32**(2) (June 1967) 198–212
14. Berger, U., Berghofer, S., Letouzey, P., Schwichtenberg, H.: Program extraction from normalization proofs. Studia Logica **82** (2005) Special issue.
15. Pierce, B.C., Steffen, M.: 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).
16. Compagnoni, A.B., Goguen, H.H.: Typed operational semantics for higher order subtyping. Information and Computation **184** (August 2003) 242–297
17. Abadi, M., Cardelli, L.: A Theory of Objects. Monographs in Computer Science. Springer Verlag (1996)
18. Odersky, M., the Scala Team: The Scala Language Specification (version 2.5). http://www.scala-lang.org/docu/files/ScalaReference.pdf (June 2007)
19. Altherr, P., Cremet, V.: Messages posted on the Scala mailing list. Accessible from http://lamp.epfl.ch/∼cremet/FGJ-omega/
20. Moors, A., Piessens, F., Joosen, W.: An object-oriented approach to datatype-generic programming. In: Workshop on Generic Programming (WGP'2006), ACM (September 2006)
21. Moors, A., Piessens, F., Odersky, M.: 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)
22. Neubauer, M., Thiemann, P.: Type classes with more higher-order polymorphism. In: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP'02), New York, NY, USA, ACM Press (2002) 179–190
23. Igarashi, A., Viroli, M.: On variance-based subtyping for parametric types. In: Proceedings of the European Conference on Object-oriented Programming (ECOOP'02), Springer-Verlag (June 2002) 441–469
24. Torgersen, M., Hansen, C.P., Ernst, E., von der Ah, P., Bracha, G., Gafter, N.: Adding wildcards to the Java programming language. In: Proceedings of the 2004 ACM symposium on Applied computing, ACM Press (2004) 1289–1296