

# Abstract Type Constructors for Java-like Languages

Philippe Altherr and Vincent Cremet

EPFL  
Switzerland

**Abstract.** Abstract type constructors, also known as type constructor polymorphism or higher-kinded types, are variables representing functions from types to types. They have been popularized by enabling the implementation of monads as a library in the functional language HASKELL. In this paper, we propose a syntax and typing rules for adding abstract type constructors to JAVA-like languages (JAVA, SCALA and C#). The proposed syntax for type constructor parameters is very compact and can be seen as a generalization of the syntax for type parameters in these languages. The syntax and typing rules are formally described. A prototype compiler let us test that soundness and decidability of the type system hold for several interesting programs but proofs of these properties are left for future work. We show that, like in HASKELL, we are able to represent monads as a library. To further demonstrate the versatility of abstract type constructors, we describe how they can be used to encode generalized algebraic data types.

## Introduction

Most mainstream programming languages let the programmer parameterize its data structures and algorithms *by types*. The general term for this mechanism is parametric polymorphism. In some contexts it is called differently: for instance, generics in JAVA [1] or templates in C++ [2]. This mechanism allows the same piece of code to be used with different type instantiations. Some languages, like HASKELL [3], additionally let the programmer parameterize its code *by type constructors*, i.e. functions from types to types. Thus, it is possible to parameterize a piece of code by a data type which is itself parameterized by a type. Although this mechanism has been widely recognized as useful, for example to represent monads [4] as a library, no attempt has been made until now, to our knowledge, to design and implement a similar feature in a JAVA-like language (JAVA, SCALA [5] and C#). In this paper we propose such a design and demonstrate its usage through some interesting examples. Our design is based on a compact notation for parameters representing type constructors. It is consistent with JAVA-like languages, both in its syntax and in its interpretation, and can be seen as a generalization of type parameters into parameters representing type constructors. It enables us to translate all HASKELL examples using abstract type constructors.

**Basic definitions.** In a first approximation, a *type constructor* is a function that takes a list of types and returns a type. For instance, the predefined SCALA operator  $\Rightarrow$  is a type constructor that takes two types  $T$  and  $U$  and returns the type of functions from  $T$  to  $U$ , noted  $T \Rightarrow U$ . Languages with parametric polymorphism let the programmer define its own type constructors: a polymorphic type `List` in such languages is actually

a type constructor which, when applied to a type  $T$ , returns the type of lists of  $T$ , noted `List[T]` in SCALA.

A language is said to support *abstract type constructors* when it provides a means of speaking of a type constructor that is only partially known. The most common approach, which is also the one that we adopted, is to provide variables that represent type constructors. The challenge is then to have a means of describing an abstract type constructor precisely enough, so that some of its features can be used, but not completely, so that it can match several instances.

*Structure of the paper.* In Section 1, we show on examples how abstract type constructors can be integrated to JAVA-like languages and what kind of programming idioms they allow. In the same time we informally introduce our notation for those parameters that represent type constructors. Because the paper has been written in the context of the SCALA project, we use a SCALA-like syntax in our design and examples, but, of course, a JAVA-like equivalent would be possible too. The rest of the paper is organized as follows: in Section 2, we justify our choices in the design of type constructor parameters. In Section 3 we present the syntax and semantics of a small calculus which is an extension of Featherweight Generic Java [6] with type constructor parameters. In Section 4, we give a complete formalization of its type system. In Section 5 we discuss pragmatic issues related to the implementation of some syntactic sugar for our extension. In Section 6, we present an original application of abstract type constructors: we interpret generalized algebraic data types (GADTs) (see [7] for a good overview) as an instance of the OO design pattern called Visitor, enhanced with abstract type constructors. Finally, we compare our work with existing work before concluding on the limitations and possible improvements of our design.

## 1 From Abstract Data Types to Abstract Type Constructors

Before introducing the concept of abstract type constructors for JAVA-like languages, we start with a short reminder on abstract data types [8] (ADTs).

### 1.1 Abstract Data Types and Object-Oriented Programming

ADTs are a powerful and well-established framework for abstraction and modularity [8]. The idea is to abstract a type by the set of operations that can be performed on it. Abstract data types can easily be modeled in an object-oriented language with generics. The solution is to have an abstract class with a type parameter representing the abstract data type and to let the operations associated with the data type be abstract members of this class. As an example, we consider the ADT representing sets of integers; it is defined by its operations: `empty`, `add`, `contains` and `fold`.

```
class SetModule[Set] {
  def empty(): Set
  def add(x: Int, xs: Set): Set
  def contains(xs: Set, x: Int): Boolean
  def fold[Y](xs: Set, f: (Int, Y) => Y, y: Y): Y

  def addAll(xs: Set, ys: Set): Set =
```

```

    this.fold[Set](xs, (x, ys) => this.add(x, ys), ys)
}

```

As exemplified by the `addAll` method above, some interesting operations can be defined on the elements of the type `Set`, without knowing its exact representation. Such generic operations will work safely with different set implementations. Here we defined `addAll` in the `SetModule` itself, but it could also have been defined separately afterwards. The resulting definition corresponds in "lifting" the current definition outside of the class `SetModule`; in the process some entities, namely `Set` and `this`, become free in the method body and must be afforded by new parameters:

```

def addAll[Set](self: SetModule[Set], xs: Set, ys: Set): Set =
  self.fold[Set](xs, (x, ys) => self.add(x, ys), ys)

```

In this encoding of ADTs, implementations of ADT are provided by subclassing the class that defines it, in this case, `SetModule`; in the process an actual type for `Set` is necessarily given. In the example below the type parameter of the `SetModule` is instantiated with the type `List[Int]`.

```

class ListSetModule extends SetModule[List[Int]] {
  def empty(): List[Int] = Nil
  def add(x: Int, xs: List[Int]): List[Int] = x :: xs
  def contains(xs: List[Int], x: Int): Boolean = xs.contains(x)
  def fold[Y](xs: List[Int], f: (Int, Y) => Y, y: Y): Y =
    xs.foldRight(y)(f)
}

```

The goal of this short introduction was to describe one way of implementing ADTs in a JAVA-like language.

## 1.2 Abstract Type Constructors

Monads have been popularized by `HASKELL` in the community of functional programming languages. Monads are used to structure computations by abstracting the way sub-computations can be composed sequentially. We could say a monad instance represents a way of sequentializing a particular kind of computation. For instance, in `HASKELL`, the `Maybe` monad describes how some computations that either succeed with one value or fail can be composed sequentially. A monad can be seen as an ADT with two operations called `unit` and `bind`. Since we just explained how to model ADTs in a JAVA-like language we could conclude that monads can be modeled as well. Unfortunately this is not the case, because what is abstract with a monad is a type constructor, i.e. a data type parameterized by a type, and not a type, as it was the case in the previous `Set` example. In the previous example we used a parameter to represent an abstract data type. For monads we would like to use a parameter to represent an abstract type constructor. This paper is about a design for adding such parameters. Here is how monads could be defined in `SCALA` if the language would allow for parameters representing type constructors.

```

class MonadModule[Monad[_]] {
  def unit[X](x: X): Monad[X]
  def bind[X,Y](xs: Monad[X], f: X => Monad[Y]): Monad[Y]

  def map[X,Y](xs: Monad[X], f: X => Y): Monad[Y] =
    this.bind[X,Y](xs, (x: X) => this.unit[Y](f(x)))
}

```

The `MonadModule` class declares a *type constructor parameter* `Monad`. All we know about this type constructor is that it expects one type argument; this is expressed by the syntax `Monad[_]`, which indicates that `Monad` represents a unary type constructor (a binary type constructor `B` would be declared with the syntax `B[_,_]`, etc). Normal type parameters are declared as usual (see for instance `X` and `Y` in method declarations).

Like in our previous set example, some generic operations, like `map`, can be defined for arbitrary monads. They work safely with different monad implementations. The class `OptionMonadModule` below is such an implementation. It describes how the SCALA class `Option`, similar to the HASKELL type `Maybe`, can be seen as a monad.

```

class OptionMonadModule extends MonadModule[Option] {
  def unit[X](x: X): Option[X] = Some(x)
  def bind[X,Y](xs: Option[X], f: X => Option[Y]): Option[Y] =
    if (xs.isEmpty()) None else f(xs.get())
}

```

### 1.3 Anonymous Type Constructors

In the previous example, the `Monad` parameter was instantiated with a class type constructor, namely `Option`. It is often the case that parameters representing type constructors are instantiated with class type constructors. In a JAVA-like language, class type constructors are *named* type constructors. But there are also cases where *anonymous* type constructors are needed. For instance, a well-known monad of the HASKELL library is the monad, called *state transformer*, that lets the programmer abstract over the modification of a global state during a sequence of computation. This monad is built from the parameterized type alias `StateT` defined below: an element of the type `StateT[S,X]` represents a computation of an element of type `X` which, as side-effect, modifies a global state of type `S`, more precisely it is a function that takes a state of type `S` and returns a pair consisting of a value of type `X` that represents the result of the computation, together with a new state.

```

type StateT[S,X] = S => Pair[X,S]

```

From this definition it is possible to define a monad for state transformers that will describe how some computations that modify a global state can be chained.

```

class StateMonadModule[S] extends MonadModule[[Z] => StateT[S,Z]] {
  def unit[X](x: X): StateT[S,X] =
    (s: S) => new Pair[X,S](x, s)
  def bind[X,Y](xs: StateT[S,X], f: X => StateT[S,Y]): StateT[S,Y] =
    (s: S) => { val p: Pair[X,S] = xs(s); f(p.fst)(p.snd) }
}

```

In this code, the expression `[Z] => StateT[S,Z]` denotes an *anonymous type constructor* that expects a type `Z` and returns the type `StateT[S,Z]`. Anonymous type constructors are similar to anonymous functions as found in functional languages, the difference being that anonymous type constructors operate on types rather than on values. In the class `MonadModule`, `unit` is declared with the return type `Monad[X]`. In the class `StateMonadModule` it is implemented with the return type `StateT[S,X]`. This is indeed correct because `StateMonadModule` instantiates `Monad` to `[Z] => StateT[S,Z]`, and `([Z] => StateT[S,Z])[X]` is equivalent to `StateT[S,X]` modulo beta-conversion.

#### 1.4 Abstract Type Constructors and OO Style

The examples presented so far are a bit unsatisfactory because they do not exploit some of the benefits of nominal object-oriented languages, of which JAVA-like languages are instances. One particularity of nominal OO languages is the 'dot' notation for selecting a method on an object. This notation gives a special status to the receiver object which enables a finer name resolution: this mechanism allows to use classes as name spaces and thus to have several times the same name as soon as they are defined in separate classes. Our formalism for type constructors is also able to exploit these benefit of OO languages.

As an example, we consider *sequences*, i.e. objects with some methods `map`, `flatMap` and `filter`. Using some simple syntactic sugar, such sequences can be elegantly defined *by comprehension*, as mathematical sets (see the SCALA for-notation [5]), but this is not important for the discussion. Using the ADT design pattern enhanced with abstract type constructors already used for monads, we can define sequences using the following abstract class.

```
class SeqModule[S[_]] {
  def map[A,B](xs: S[A], f: A => B): S[B]
  def flatMap[A,B](xs: S[A], f: A => S[B]): S[B]
  def filter[A](xs: S[A], f: A => Boolean): S[A]
}
```

What is not convenient with this approach is that, given a particular integer sequence object `xs`, we must use an explicit `SeqModule sm` each time we want to call one of the sequence functions. In short, instead of writing `sm.filter[Int](xs, test)` we would prefer to write `xs.filter(test)`, i.e. to select the method on the first argument of the function. In the latter solution, we save two annotations: `sm` and the type parameter `Int`. In languages with inference of type arguments, like SCALA and JAVA 1.5, the latter could be omitted but not the other.

The idea is to generalize an old trick for representing binary methods, that relies on a type parameter that is bound by a type instantiated with itself. But let us explain this more precisely. There is an OO design pattern based on a recursive F-bounded polymorphism [9], that aims at defining a method such that one of its parameters is of the same type as the current instance (also called binary method). For instance, suppose we want to define an interface for all ordered types. One well-known solution consists in writing the following class.

```

class Order[X <: Order[X]] {
  def lt(that: X): Boolean

  def ge(that: X): Boolean = !(this.lt(that))
}

class Integer extends Order[Integer] {
  val x: Int
  def lt(that: Integer): Boolean = this.x < that.x
}

```

The same trick can be used for representing sequences. With `Order` we had a type parameter bound by a class type instantiated with itself. Similarly we will now use a type constructor parameter bound by a type containing itself.

```

class Seq[A, S[X] <: Seq[X,S]] {
  def map[B](f: A => B): S[B]
  def flatMap[B](f: A => S[B]): S[B]
  def filter(f: A => Boolean): S[A]
}

```

The first parameter `A` of the class `Seq` is a type. It represents the type of the elements of the sequence. The second parameter `S` is a type constructor with one parameter `X`. This type constructor represents the data type that implements the sequence. Here, unlike in the definition of `SeqModule`, its type parameter has been named. This was done in order to specify its bound `Seq[X,S]`. If that bound was omitted, we could have used the syntax `S[_]`. However, if `S` had no bound, an expression like `xs.map(f).filter(p)` would not be well-formed. Indeed, the result of the call to `map` would be some data type of which nothing is known and the call to `filter` would therefore not be allowed. If the bound is present, we know that the data type inherits from `Seq`, which defines a method `filter`.

The canonical example of a sequence is a list:

```

class List[A] extends Seq[A,List] {
  def map[B](f: A => B): List[B] = ...
  def flatMap[B](f: A => List[B]): List[B] = ...
  def filter(f: A => Boolean): List[A] = ...
}

```

## 2 Type Constructor Parameters: Syntax and Interpretation

The key of our smooth integration of abstract type constructors in JAVA-like languages is a compact syntax for type constructor parameters. In this section we explain this syntax and its interpretation. For a formal account on the concepts presented here, the reader is invited to wait until the next section.

This section is structured as follows. First, we remind what characterizes class type constructors, as found in JAVA-like languages, and as formalized in FGJ. Then, we propose a syntax for type constructor parameters that is similar to the syntax of class type constructors in FGJ. In the process, we observe that type constructor parameters generalize type parameters, which can be seen as type constructor parameters of arity 0.

## 2.1 About Class Type Constructors in FGJ

**Class type constructors.** The declaration of a class  $C$  in FGJ has multiple effects, one of them is to provide a new *type constructor*, also named  $C$ .  $C$  alone is not considered as a type, however it can become a type if it is fed with a sufficient number of type arguments  $\bar{T}$ . Types built this way have the form  $C[\bar{T}]$  and are called *class types* because they refer to a class  $C$ . Since the symbol  $C$  is used for building class types, we call it a *class type constructor*. In FGJ, if a type is not a class type it can only be a type variable  $X$ . Contrary to class symbols  $C$ , type variable symbols  $X$  represent directly a type.

In JAVA-like languages, we can define a class without type parameters and using it directly as a type, without writing square brackets. This greatly alleviates the syntax but theoretically it is easier to think of classes without type parameters as classes that take an empty section of type parameters. The following code

```
class A { val x: A  val y: List[A] }
```

is then seen as syntactic sugar for

```
class A[] { val x: A[]  val y: List[A[]] }
```

**Sections of type parameters.** In JAVA-like languages, type parameters are grouped in *sections* and such sections must be interpreted as a whole. For instance the following class `Pair` declares a section with two type parameters, `X <: Object` and `Y <: Object`.

```
class Pair[X <: Object, Y <: Object]
  val fst: X
  val snd: Y
}
```

What prevents us of interpreting type parameters separately is that the bound of one type parameter can in principle depend on another type parameter. This is the case in the following program where the bound `A[Y]` of the type parameter `Z` refers to the other type parameter `Y`.

```
class A[X <: Object]
class B[Y <: Object, Z <: A[Y]]
```

**Domain of a type constructor.** A mathematical function is characterized by a set of input values (its domain) and a set of output values (its range). Type constructors are functions from types to types and inherit of the concepts of domain and range. In type theory the combination of the domain and the range of a type constructor constitutes its *kind*.

We can find the same concepts in FGJ. A class type constructor is defined on a limited type *domain* determined by the bounds of its type parameters. For instance by declaring the previous class for pairs, we implicitly constrain the domain  $\mathcal{D}$  of the type constructor `Pair` to be all tuples of types  $(T, U)$  such that  $T$  and  $U$  are subtypes of `Object`. Such a set  $\mathcal{D}$  is called *the domain of the type constructor Pair*.

The domain of a class type constructor  $C$  is entirely determined by the section of type parameters of its associated class declaration. The result of applying a class type constructor  $C$  to a tuple of types is always a type and never another type constructor, so the set of all types is a first approximation of the range of a class type constructor.

**Parametric subtyping and range of a class type constructor.** By declaring, in JAVA-like languages, that a class B has a superclass A we implicitly define a subtyping relation between the class types A and B, namely that B is a subtype of A. Now, if the class B has a type parameter X, the subtyping relation between A and B is implicitly parameterized by all elements of the domain of B, as if the type parameter X was universally quantified. For instance, considering the following example program, there is a subtyping rule in FGJ, called (S-CLASS), that lets us derive that B[T] is a subtype of A[T,T] for all types T

```
class A[X <: Object, Y <: Object]
class B[X <: Object] extends A[X,X]
```

In other words, the set of types that can be reached by applying a class type constructor  $C$  is constrained by the superclass of its associated class. With this observation we can refine the definition of *the range of a class type constructor*: the range of a class type constructor  $C$  with type parameters  $\bar{X}$  and superclass  $N$  is the set of types  $T$  such that  $T$  is a subtype of  $N\{\bar{X}\setminus(\bar{T})\}$  for some tuple  $(\bar{T})$  belonging to the domain of  $C$ .

**Summary.** A class declaration defines a class type constructor. Type parameters are grouped in sections. Class type constructors have a domain determined by the bounds of their type parameters and a range determined by their superclass, which is parametrically interpreted as a supertype. All class type constructors, without exception, expect a section of type parameters (potentially implicit if empty). A class declaration is thus a very compact notation for describing a large amount of information about the associated class type constructor. Our design for type constructor parameters adopts the same philosophy and a similar syntax.

## 2.2 About Type Constructor Parameters

We just said that, in FGJ, all the information about a class type constructor is contained in its symbol  $C$ , its section of type parameters  $\bar{P}$  and its superclass  $N$ . For type constructor parameters, we make the same assumptions: all the information about a type constructor parameter is contained in its symbol  $X$ , its section of type parameters  $\bar{P}$  and its bound  $N$ . We adopt the abstract syntax  $X[\bar{P}] <: N$  for grouping these pieces of information.

**Minimal domain and maximal range.** The interpretation of the construct  $X[\bar{P}] <: N$  is identical to the interpretation of an hypothetical class type constructor  $X$  with a section of type parameters  $\bar{P}$  and a superclass  $N$ : the domain of  $X$  is determined by the parameters  $\bar{P}$  and the range of  $X$  is determined by  $N$ . There is only one subtlety in the interpretation, that is due to the abstract nature of parameters: for a type constructor parameter  $X$ , the parameters  $\bar{P}$  represents a *minimal* domain whereas for a class type constructor they represent an *exact* domain. By minimal, we mean that a valid instance for the parameter  $X$  is a type constructor whose domain *contains* the domain represented by  $\bar{P}$  and not necessarily that *equals* it. Symmetrically, the bound  $N$  of a type constructor parameter represents a *maximal* range whereas for a class type constructor it represents an *exact* range. By maximal, we mean that a valid instance for

the parameter  $X$  is a type constructor whose range is *included in* the range represented by  $N$  and not necessarily that *equals* it.

The observation about type constructor parameters on the minimality of their domain and the maximality of their range has a link with the subtyping of functional types: functional types can be made contravariant (resp. covariant) and not necessarily invariant in the type of the arguments (resp. the result). This is not surprising if we establish the following correspondences: type constructors are functions (from types to types), and type constructor parameters represent the types of these functions.

**Example of interpretation.** In the example of sequences introduced in Section 1.4, the class `Seq` declares a type constructor parameter `S[X] <: Seq[X,S]`. The declaration contains some syntactic sugar. First, every parameter without type bound is implicitly bound by the class type `Object[]`. Then, as for class type constructors, we always require a type constructor parameter to have a section of parameters at the level of the abstract syntax (we explain later in Section 5 how to deal in practice with parameterless type constructor parameters). Thus, the complete, desugared, version of the previous parameter declaration is:

$$S[X[] <: Object[]] <: Seq[X,S]$$

While short, this declaration contains actually a lot of information. It is equivalent to the following English prose:

*"We declare a unary type constructor parameter named S. The domain of S contains the set D of type constructors X<sub>1</sub> taking no arguments such that X<sub>1</sub>[] is a subtype of Object[]. Finally, S must satisfy the property that S[X<sub>2</sub>] is a subtype of Seq[X<sub>2</sub>,S] for all element X<sub>2</sub> in D".*

Note that the parameter `S` being defined appears in its bound `Seq[X,S]`. Note also that the domain of a type constructor parameter is syntactically expressed by a section of type constructor parameters; 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 section declaration, the name `X` is a binder that serves simultaneously two purposes, defining the domain of `S` and defining its range. In the English description of the parameter `S` we made this double role explicit by using a different variable for each role, namely  $X_1$  and  $X_2$ .

**Generalization of type parameters.** An important characteristic of our design is that there is no concept of type parameters any longer. Methods and classes take as arguments type constructor parameters only. This choice is justified because type parameters can always be represented with type constructor parameters of arity 0: a FGJ type parameter declaration  $X <: N$  is equivalent to  $X[] <: N$  in our design. In practice it is nevertheless convenient to let type parameters and type constructor parameters coexist because a type variable `X` is more user-friendly than the verbose form `X[]`.

**Conclusion.** We have seen that, in our proposal, type constructor parameters are similar in syntax and interpretation to the declaration of class type constructors, in particular they inherit from their compact and expressive notation. The good point is

that type constructor parameters are also a generalization of type parameters. These two clear relations with FGJ are important for supporting the claim that our extension of FGJ is indeed natural.

### 3 FGJ<sub>ω</sub>

Featherweight Generic Java (FGJ) is a simple nominal object-oriented calculus where both classes and methods are parameterized by types. In this section we present the syntax and semantics of an extension of FGJ with abstract type constructors. Since our extension generalizes FGJ’s type parameters by parameters representing type constructors, the resulting calculus is called FGJ<sub>ω</sub> (pronounce "FGJ-omega"), by analogy with λ<sub>ω</sub>, the simply typed lambda-calculus extended with type operators.

We discuss the interesting aspects of the formalization by focusing on the points that are different in FGJ and FGJ<sub>ω</sub>. A completely rigorous formalization, using De Bruijn’s indices [10] instead of variables, has been developed in the COQ proof assistant [11] and can be found in [12].

#### 3.1 Syntax

The abstract syntax of FGJ<sub>ω</sub> is given in Figure 1. Compared to FGJ, abstract type constructors have been added and type casts and method overriding have been removed.

*General notations.* Sequences are ubiquitous in our formalism. We write  $\emptyset$  for an empty sequence of elements. We write  $|s|$  for the length of a sequence  $s$ . We write  $\bar{x}$  for a finite sequence of elements ranged over by the meta-variable  $x$ . We also adopt the convention of using the meta-variable  $x_{opt}$  for representing an optional element of the set denoted by a meta-variable  $x$ .

A FGJ<sub>ω</sub> program consists of a list  $\bar{D}$  of class declarations and a main term  $t$  to be evaluated in the context of these classes. A class declaration is characterized by its unique name  $C$ , a section of type constructor parameters  $\bar{P}$ , an optional superclass  $N_{opt}$  and a list of members  $\bar{M}$ .

A member is either a field declaration, a method declaration or a method implementation. As in FGJ, the implementation of fields is deferred until instances are created, which explains why there is no member corresponding to a field implementation.

Contrary to FGJ we make a distinction between the declaration and the implementation of a method. A method declaration defines a new method symbol  $m$  and contains the declaration of its parameters and of its result type  $T$ . The parameters of a method are composed of type constructor parameters, grouped in a section  $\bar{P}$ , and of value parameters  $\bar{x} : \bar{T}$ . A method implementation `def  $m = t$`  provides a body  $t$  for a method  $m$  that has been declared in the current class or one of its superclasses. A particularity of our calculus is that the scope of method parameters extends over the body of every implementation of the method, hence the absence of parameters in a method implementation; this simplifies the theory and the implementation since they do not have to deal with parameter symbols that are declared at different locations but that represent the same entities. Note nevertheless that to ease the reading of our examples, we copy the complete signature of a method in every method implementation, but that has to be considered only as syntactic sugar.

The terms of our calculus are similar to the ones of FGJ: there are constructs for representing variables, field selections, method calls and instance creations. As in FGJ we represent the current instance of a class by a variable, which is named `this` by convention. The only difference is in the construct `new N{val  $\bar{f}$  =  $\bar{t}$ }` for creating new class instances: in both formalisms, all fields of the object being created must be given a value at this point, but  $\text{FGJ}_\omega$  uses the names  $\bar{f}$  of the fields for making the association field/value whereas FGJ uses their position.

The definitions of types, type constructors and type constructor parameters are mutually recursive.

A parameter  $X[\bar{P}] <: N$  declares a type constructor  $X$ . The minimal domain of  $X$ , i.e. the set of arguments it must at least be able to accept, is itself described by a sequence  $\bar{P}$  of type constructor parameters. Finally,  $N$ , the bound of  $X$ , represents the maximal range of  $X$ : for every tuple  $(\bar{K})$  of type constructors in the minimal domain of  $X$ ,  $X[\bar{K}]$  must be a subtype of  $N\{\text{vars}(\bar{P}) \setminus (\bar{K})\}$ . In this expression,  $\text{vars}(\bar{P})$  denotes the parameters declared in  $\bar{P}$ , they can appear in  $N$  and must be substituted by the  $\bar{K}$ . More generally, an expression  $T\{\bar{X} \setminus (\bar{K})\}$  denotes the simultaneous substitution in  $T$  of the type constructors  $\bar{K}$  for the variables  $\bar{X}$ .

A type constructor is either a variable  $X$ , corresponding to a type constructor parameter, a class type constructor  $C$ , corresponding to a class declaration, or an anonymous type constructor  $[\bar{P}] \Rightarrow T$  that expects arguments of kind  $\bar{P}$  and that returns the type  $T$ . This last construct is analogous to anonymous functions in functional programming languages (or lambda-abstractions in the lambda-calculus) except it operates at the level of types and not at the level of values. It must not be confused with the arrow notation  $T \Rightarrow U$  that we sometimes use in the examples for representing functional types.

Finally, there is only one construct for types: a type  $K[\bar{K}]$  is made of a type constructor  $K$  applied to a list of type constructors  $\bar{K}$ . This construct subsumes both FGJ class types and type variables: if  $K$  is a class symbol  $C$  we get a class type, if  $K$  is a parameter symbol  $X$  and that the sequence  $\bar{K}$  is empty we get the type  $X[]$  which represents an abstract type. Of course, we can also represent types that are not expressible in FGJ: if  $X$  is a variable representing a unary type constructor (for instance `Monad`) and  $C$  is a parameterless class symbol (for instance `Integer`), we can form the type  $X[C]$ .

**Abbreviations.** In the abstract syntax, we assume that all symbols used in a program have been declared somewhere and that all declarations define different names. In the rules, we often use premises of the form  $(\text{val } f : T) \in C$  for specifying that the field  $f$  is declared in class  $C$  with type  $T$ ; in this expression, the presence of the program  $p$  we are talking about is implicit. We use similar abbreviations for classes and methods. All these abbreviations can be understood from the syntax of  $\text{FGJ}_\omega$ .

### 3.2 Semantics

As for FGJ, the computational meaning of a  $\text{FGJ}_\omega$  program is defined by a small-step operational semantics. We additionally enforce a call-by-value strategy. Figure 3 contains the definition of a one-step reduction relation. The full evaluation of a program  $p$  then consists in repeatedly reducing the main term  $t$  of the program in the context of its class declarations  $\bar{D}$  until a value  $v$  is reached. Values are the subset of terms that

Class name	$A, B, C$	
Method name	$m$	
Field name	$f$	
Variable	$x$	$::= \text{this} \mid \dots$
Type constructor variable	$X, Y$	
Class	$D$	$::= \text{class } C[\overline{P}] \text{ extends } N_{opt} \{ \overline{M} \}$
Member	$M$	$::= \text{val } f : T$ field declaration $\mid \text{def } m[\overline{P}](\overline{x} : \overline{T}) : T$ method declaration $\mid \text{def } m = t$ method implementation
Term	$t, u$	$::= x$ variable, current instance $\mid t.f$ field selection $\mid t.m[\overline{K}](\overline{t})$ method call $\mid \text{new } N\{\text{val } \overline{f} = \overline{t}\}$ instance creation
Type constr. parameter	$P, Q$	$::= X[\overline{P}] <: N$
Type constructor	$K$	$::= X$ type constructor variable $\mid C$ class type constructor $\mid [\overline{P}] \Rightarrow T$ anonymous type constructor
Type	$S, T, U$	$::= K[\overline{K}]$
Class type	$N$	$::= C[\overline{K}]$
Program	$p$	$::= \overline{D} t$
Value	$v, w$	$::= \text{new } N\{\text{val } \overline{f} = \overline{v}\}$
Evaluation context	$e$	$::= \langle \rangle \mid e.f \mid e.m[\overline{K}](\overline{t}) \mid v.m[\overline{K}](\overline{v}, e, \overline{t})$ $\mid \text{new } N\{\text{val } \overline{f} = \overline{v}, e, \overline{t}\}$

Fig. 1. Syntax of FGJ<sub>ω</sub>

have the form  $\text{new } N\{\text{val } \overline{f} = \overline{v}\}$ . They correspond to objects with fully evaluated fields.

The definition of the one-step reduction relation needs a subtyping relation between class types, summarized in Figure 2. Two class types  $C[\overline{K}]$  and  $C'[\overline{K}']$  are in this relation if  $C$  is a subclass of  $C'$  and the arguments  $\overline{K}'$  of  $C'$  are obtained by propagating the arguments  $\overline{K}$  of  $C$  through the class hierarchy.

There are three rules for reducing a term. The rules (R-SELECT) and (R-CONTEXT) do not need additional explanations. The rule (R-CALL) for reducing a method call  $v.m[\overline{K}](\overline{v})$  is a bit more complex: first, we recover the parameters  $\overline{X}$  and  $\overline{x}$  of the method  $m$ , then we compute the arguments  $\overline{K}'$  of the enclosing class  $C$  of the method implementation as seen from the receiver object  $v$ , finally we perform all the needed substitutions. Note that the substitution of a value  $v$  for a variable  $x$  in a term  $t$  is noted  $t\{x \setminus v\}$ .

$\text{(SC-REFL)} \frac{}{C \triangleleft C}$ $\text{(SC-EXTENDS)} \frac{\text{class } C[\bar{P}] \text{ extends } C'[\bar{K}] \{ \bar{M} \} \quad C' \triangleleft C''}{C \triangleleft C''}$	$\text{(CS-REFL)} \frac{}{N \triangleleft : N}$ $\text{(CS-EXTENDS)} \frac{\text{class } C[\bar{P}] \text{ extends } N \{ \bar{M} \} \quad \text{vars}(\bar{P}) = \bar{X} \quad N\{\bar{X} \setminus \bar{K}\} \triangleleft : N'}{C[\bar{K}] \triangleleft : N'}$
--	--

**Fig. 2.** Subclassing ( $C \triangleleft C'$ ) and class type subtyping ( $N \triangleleft : N'$ )

$\text{(R-SELECT)} \frac{v = \text{new } N \{ \text{val } \bar{f} = \bar{v} \}}{v.f_i \rightarrow v_i}$	$\text{(R-CALL)} \frac{\text{def } m[\bar{P}](\bar{x} : \bar{T}) : T \quad \text{vars}(\bar{P}) = \bar{X} \quad v = \text{new } N \{ \text{val } \bar{f} = \bar{w} \} \quad N \triangleleft : C[\bar{K}'] \quad (\text{def } m = t) \in C \quad \text{class } C[\bar{Q}] \text{ extends } N_{opt} \{ \bar{M} \} \quad \text{vars}(\bar{Q}) = \bar{Y}}{v.m[\bar{K}](\bar{v}) \rightarrow t\{\bar{X} \setminus \bar{K}\}\{\bar{Y} \setminus \bar{K}'\}\{\text{this} \setminus v\}\{\bar{x} \setminus \bar{v}\}}$
$\text{(R-CONTEXT)} \frac{t \rightarrow t'}{e\langle t \rangle \rightarrow e\langle t' \rangle}$	

**Fig. 3.** Computation ( $t \rightarrow u$ )

## 4 Type System

The rules that define the well-formedness of a program are divided into several typing judgments. Most of them are parameterized by a typing context. Typing judgments that only deal with types, like subtyping for instance, need a typing context for types. The judgment that assigns a type to a term additionally needs a typing context for terms. A typing context for types  $\Delta$  is a sequence of parameter sections  $\{\bar{P}\}, \dots, \{\bar{Q}\}$ , and a typing context for terms  $T$  is a sequence of bindings  $\bar{x} : \bar{T}$  that associate types to variables. The rest of this section is dedicated to the explanation of the more interesting typing judgments. The formalization uses two auxiliary functions, defined in Figure 4, that resp. return the bound of a type and the parameters of a type constructor.

### 4.1 Subtyping

Subtyping in  $\text{FGJ}_\omega$  (see Figure 5) resembles subtyping in  $\text{FGJ}$ . In particular, rules (S-REFL), (S-TRANS) and (S-CLASS) are identical in both formalisms. Rule (S-VAR) is generalized in  $\text{FGJ}_\omega$  in order to accommodate the fact that type constructor variables have arguments that must be propagated: where the supertype of a type variable  $X$  is directly its declared bound  $N$  in  $\text{FGJ}$ , the supertype of a variable type  $X[\bar{K}]$  is computed from  $N$  by replacing the parameters  $\bar{X}$  of  $X$  with its arguments  $\bar{K}$  in  $\text{FGJ}_\omega$ .

$\text{(B-CLASS)} \frac{}{\text{bound}_\Delta(C[\overline{K}]) = C[\overline{K}]}$ $\text{(B-VAR)} \frac{\{\overline{P}\} \in \Delta \quad X[\overline{Q}] <: N \in \overline{P} \quad \text{vars}(\overline{Q}) = \overline{X}}{\text{bound}_\Delta(X[\overline{K}]) = N\{\overline{X}\} \setminus \{\overline{K}\}}$ $\text{(B-FUN)} \frac{\text{vars}(\overline{P}) = \overline{X} \quad \text{bound}_\Delta(T\{\overline{X}\} \setminus \{\overline{K}\}) = N}{\text{bound}_\Delta([\overline{P}] \Rightarrow T)[\overline{K}] = N}$	$\text{(P-CLASS)} \frac{\text{class } C[\overline{P}] \text{ extends } N_{opt} \{ \overline{M} \}}{\text{params}_\Delta(C) = \overline{P}}$ $\text{(P-VAR)} \frac{\{\overline{P}\} \in \Delta \quad (X[\overline{Q}] <: N) \in \overline{P}}{\text{params}_\Delta(X) = \overline{Q}}$ $\text{(P-FUN)} \frac{}{\text{params}_\Delta([\overline{P}] \Rightarrow T) = \overline{P}}$
--	---

**Fig. 4.** Bound ( $\text{bound}_\Delta(T) = N$ ) and parameters ( $\text{params}_\Delta(K) = \overline{P}$ )

The rule (S-BETA) does not exist in FGJ, it integrates *beta-conversion* in the subtyping relation.

$\text{(S-REFL)} \frac{}{\Delta \vdash T <: T}$
$\text{(S-TRANS)} \frac{\Delta \vdash T <: S \quad \Delta \vdash S <: U}{\Delta \vdash T <: U}$
$\text{(S-VAR)} \frac{\{\overline{P}\} \in \Delta \quad (X[\overline{Q}] <: N) \in \overline{P} \quad \text{vars}(\overline{Q}) = \overline{X}}{\Delta \vdash X[\overline{K}] <: N\{\overline{X}\} \setminus \{\overline{K}\}}$
$\text{(S-CLASS)} \frac{\text{class } C[\overline{P}] \text{ extends } N \{ \overline{M} \} \quad \text{vars}(\overline{P}) = \overline{X}}{\Delta \vdash C[\overline{K}] <: N\{\overline{X}\} \setminus \{\overline{K}\}}$
$\text{(S-BETA)} \frac{T =_\beta U}{\Delta \vdash T <: U}$

**Fig. 5.** Subtyping ( $\Delta \vdash T <: U$ )

**Beta-conversion.** Our type system identifies types that are equivalent modulo beta-conversion. By definition, two types  $T$  and  $U$  are beta-convertible, noted  $T =_\beta U$ , if it is possible to go from one type to the other by applying the beta-rule below, potentially multiple times, at any depth inside the type, and in one direction or the other.

$$\text{(BETA)} \frac{\text{vars}(\overline{P}) = \overline{X}}{([\overline{P}] \Rightarrow T)[\overline{K}] =_\beta T\{\overline{X}\} \setminus \{\overline{K}\}}$$

In the beta-rule, the left hand-side is a direct application of an anonymous type constructor, such a type expression is called a beta-redex. In practice, since we can prove

that the application of the beta-rule is terminating and confluent on well-formed types (well-formed types correspond to the terms of the simply typed lambda-calculus [13], which is terminating and confluent) we can just compare the beta-normal forms of two types in order to see if they are beta-convertible.

Beta-conversion is used in the subtyping rule (S-BETA). It lets us identify types like  $([X] \Rightarrow \text{List}[X])[\text{Int}]$  and  $\text{List}[\text{Int}]$ . In this case, the beta-redex is at the top-level. One natural question arises: do we need in practice to consider inner redexes, i.e. redexes that are not at the top-level? The answer is yes since it can be useful to write types like  $\text{List}[\text{Monad}[\text{Int}]]$  where `Monad` is a variable. Since `Monad` can in principle be instantiated by an anonymous type constructor, as in our example of the monad transformer in Section 1.3, we end up with a type  $\text{List}[(X \Rightarrow \text{StateT}[S, X])[\text{Int}]]$  that must be identified, by deep beta-conversion, with the type  $\text{List}[\text{StateT}[S, \text{Int}]]$ . An example of code that exhibits a type like  $\text{List}[\text{Monad}[\text{Int}]]$  is the `sequence` function below, which can be defined in the class `MonadModule`. It transforms a list of monads into a monad of lists.

```
def sequence[X](xs: List[Monad[X]]): Monad[List[X]] =
  if (xs.isEmpty())
    this.unit[List[X]](Nil)
  else
    this.bind[X, List[X]](xs.head(),
      (x: X) =>
        this.bind[List[X], List[X]](this.sequence[X](xs.tail()),
          (xs: List[X]) =>
            this.unit[List[X]](x :: xs)))
```

**Eta-conversion.** When designing our calculus, we eventually had to face the following question: should the type constructors `String` and  $[\ ] \Rightarrow \text{String}[\ ]$  be considered equivalent? In other words, should the equality between type constructors also include eta-conversion? Finally we decided not to consider eta-conversion and to let to the programmer the responsibility of writing types in a manner it is never necessary to compare type constructors like above. In this choice we follow the philosophy of the calculus  $F_{<}^{\omega}$ : with type operators and subtyping, which is described in Chapter 31 of Pierce’s TPL book [14].

## 4.2 Well-formedness of Types and Type Constructors

The definitions of well-formedness for types, type constructors and type constructor parameters are mutually recursive and given in Figure 6. Most of the rules are straightforward. The interesting rule is the rule (K-APPLY) that checks the well-formedness of a type  $K[\overline{K}]$ . After a routine recursive check that the type constructors  $K$  and  $\overline{K}$  are well-formed, the rule must check that the arguments  $\overline{K}$  of  $K$  conform to its expected parameters  $\overline{P}$ . This action is performed by an auxiliary relation,  $\Delta \vdash (\overline{K}) \in \{\overline{P}\}$ , which is explained below.

**Conformance of type constructors.** Figure 7 explains under which conditions a section  $\overline{K}$  of type constructors satisfies the requirements expressed by a section

$$\begin{array}{c}
\text{(K-CLASS)} \frac{}{\Delta \vdash C \text{ WF}} \\
\text{(K-VAR)} \frac{\{\bar{P}\} \in \Delta \quad X \in \text{vars}(\bar{P})}{\Delta \vdash X \text{ WF}} \\
\text{(K-FUN)} \frac{\Delta, \{\bar{P}\} \vdash \bar{P} \text{ WF} \quad \Delta, \{\bar{P}\} \vdash T \text{ WF}}{\Delta \vdash [\bar{P}] \Rightarrow T \text{ WF}} \\
\text{(K-APPLY)} \frac{\Delta \vdash K \text{ WF} \quad \Delta \vdash \bar{K} \text{ WF} \quad \text{params}_{\Delta}(K) = \bar{P} \quad \Delta \vdash (\bar{K}) \in \{\bar{P}\}}{\Delta \vdash K[\bar{K}] \text{ WF}} \\
\text{(K-PARAM)} \frac{\Delta, \{\bar{P}\} \vdash \bar{P} \text{ WF} \quad \Delta, \{\bar{P}\} \vdash N \text{ WF}}{\Delta \vdash X[\bar{P}] <: N \text{ WF}}
\end{array}$$

**Fig. 6.** Type well-formedness ( $\Delta \vdash K \text{ WF}$ ,  $\Delta \vdash T \text{ WF}$ ,  $\Delta \vdash P \text{ WF}$ )

$\bar{P}$  of type constructor parameters. The corresponding judgment can be understood as the membership of the tuple  $(\bar{K})$  to a set denoted by the section  $\bar{P}$ , hence the notation  $\Delta \vdash (\bar{K}) \in \{\bar{P}\}$  for this judgment. For testing the conformance (see rule (SATISFACTION)), a simultaneous substitution of the  $\bar{K}$  for the variables  $\bar{X}$  declared by  $\bar{P}$  is first performed in the  $\bar{P}$ . Then, an auxiliary relation that checks a one-to-one conformance is called for every element of  $\bar{K}$ ; this relation is defined by a unique rule called (MODELS).

In the premises of the rule (MODELS), we morally want to check that the domain of  $K$ , represented by  $\bar{Q}$ , contains the domain denoted by the section  $\bar{P}$  (contravariance check). This condition is mathematically equivalent to the logical statement  $\forall \bar{X} \in \{\bar{P}\}, (\bar{X}) \in \{\bar{Q}\}$ , which is implemented by the judgment  $\Delta, \{\bar{P}\} \vdash (\bar{X}) \in \{\bar{Q}\}$ , in the premises of the rule. The other premise, namely  $\Delta, \{\bar{P}\} \vdash K[\bar{X}] <: N$  (covariance check), checks that the range of  $K$  conforms to the expected range for  $X$ : such a check is equivalent to the logical statement  $\forall \bar{X} \in \{\bar{P}\}, K[\bar{X}] <: N$ .

$$\begin{array}{c}
\text{(SATISFACTION)} \frac{\text{vars}(\bar{P}) = \bar{X} \quad \Delta \models \bar{K} : \bar{P}\{(\bar{X}) \setminus (\bar{K})\}}{\Delta \vdash (\bar{K}) \in \{\bar{P}\}} \\
\text{(MODELS)} \frac{\text{vars}(\bar{P}) = \bar{X} \quad \text{params}_{\Delta}(K) = \bar{Q} \quad \Delta, \{\bar{P}\} \vdash (\bar{X}) \in \{\bar{Q}\} \quad \Delta, \{\bar{P}\} \vdash K[\bar{X}] <: N}{\Delta \models K : X[\bar{P}] <: N}
\end{array}$$

**Fig. 7.** Satisfaction ( $\Delta \vdash (\bar{K}) \in \{\bar{P}\}$ ,  $\Delta \models K : P$ )

### 4.3 Typing of Terms

The typing of terms, given in Figure 8, is in the spirit of FGJ. We just mention two interesting points. The first one is the use of the conformance check  $\Delta \vdash (\bar{K}) \in \{\bar{P}\{(\bar{Y})\backslash(\bar{K}')\}\}$  in the rule (T-CALL) for testing that the actual arguments  $\bar{K}$  of the method conform to the expected parameters  $\bar{P}$ . The other point is the use of an auxiliary relation  $\text{isComplete}(C, \bar{f})$  in rule (T-NEW) for checking that the instance being created does not contain unimplemented fields or methods.

$\text{(T-VAR)} \frac{x : T \in \Gamma}{\Delta; \Gamma \vdash x : T}$
$\text{(T-SELECT)} \frac{\Delta; \Gamma \vdash t : S \quad \text{bound}_\Delta(S) = N \quad \text{params}_\Delta(C) = \bar{P} \quad \text{vars}(\bar{P}) = \bar{X} \quad N \triangleleft : C[\bar{K}]}{\Delta; \Gamma \vdash t.f : T\{\bar{X}\}\backslash(\bar{K})}$
$\text{(T-CALL)} \frac{\Delta; \Gamma \vdash t : S \quad \Delta \vdash \bar{K} \text{ WF} \quad \Delta; \Gamma \vdash \bar{t} : \bar{S} \quad \text{bound}_\Delta(S) = N \quad N \triangleleft : C[\bar{K}'] \quad (\text{def } m[\bar{P}](\bar{x} : \bar{T}) : T) \in C \quad \text{params}_\Delta(C) = \bar{Q} \quad \text{vars}(\bar{Q}) = \bar{Y}}{\Delta \vdash (\bar{K}) \in \{\bar{P}\{(\bar{Y})\}\backslash(\bar{K}')\}} \quad \Delta \vdash \bar{S} <: \bar{T}\{\bar{X}\}\backslash(\bar{K})\{(\bar{Y})\}\backslash(\bar{K}')}{\Delta; \Gamma \vdash t.m[\bar{K}](\bar{t}) : T\{\bar{X}\}\backslash(\bar{K})\{(\bar{Y})\}\backslash(\bar{K}')}$
$\text{(T-NEW)} \frac{\Delta \vdash N \text{ WF} \quad \Delta; \Gamma; N \vdash \text{val } \bar{f} = \bar{t} \text{ WF} \quad N = C[\bar{K}] \quad \text{isComplete}(C, \bar{f})}{\Delta; \Gamma \vdash \text{new } N\{\text{val } \bar{f} = \bar{t}\} : N}$
$\text{(WF-FIELD-VAL)} \frac{\Delta; \Gamma \vdash t : S \quad N \triangleleft : C[\bar{K}] \quad (\text{val } f : T) \in C \quad \text{params}_\Delta(C) = \bar{P} \quad \text{vars}(\bar{P}) = \bar{X} \quad \Delta \vdash S <: T\{\bar{X}\}\backslash(\bar{K})}{\Delta; \Gamma; N \vdash \text{val } f = t \text{ WF}}$
$\text{(NEW-COMPLETE)} \frac{(1) \forall C', f, T. \quad C \triangleleft C' \wedge (\text{val } f : T) \in C' \Rightarrow \exists i. \quad f_i \equiv f \quad (2) \forall C', m, \bar{T}, T. \quad C \triangleleft C' \wedge (\text{def } m[\bar{P}](\bar{T}) : T) \in C' \Rightarrow \exists A, t. \quad C \triangleleft A \wedge (\text{def } m = t) \in A}{\text{isComplete}(C, \bar{f})}$

**Fig. 8.** Typing ( $\Delta; \Gamma \vdash t : T$ )

### 4.4 Program well-formedness

A program  $p$  is considered well-formed when all the following properties are satisfied. There are no cycles in the class hierarchy. All classes are well-formed. The main term is

well-typed in the empty context. And finally, a method implementation is never overridden in a subclass. The well-formedness relation for members and classes is defined in Figure 9.

$$\begin{array}{c}
\text{(WF-FIELD-DECL)} \frac{\text{class } C[\bar{Q}] \text{ extends } N_{opt} \{ \bar{M} \} \quad \Delta = \{ \bar{Q} \} \quad \Delta \vdash T \text{ WF}}{C \vdash \text{val } f : T \text{ WF}} \\
\text{(WF-METHOD-DECL)} \frac{\text{class } C[\bar{Q}] \text{ extends } N_{opt} \{ \bar{M} \} \quad \Delta = \{ \bar{Q} \}, \{ \bar{P} \} \quad \Delta \vdash \bar{P} \text{ WF} \quad \Delta \vdash \bar{T} \text{ WF} \quad \Delta \vdash T \text{ WF}}{C \vdash \text{def } m[\bar{P}](\bar{x} : \bar{T}) : T \text{ WF}} \\
\text{(WF-METHOD-IMPL)} \frac{\begin{array}{l} \text{class } C[\bar{Q}] \text{ extends } N_{opt} \{ \bar{M} \} \quad \text{vars}(\bar{Q}) = \bar{Y} \\ \text{class } C'[\bar{Q}'] \text{ extends } N'_{opt} \{ \bar{M}' \} \quad \text{vars}(\bar{Q}') = \bar{Y}' \\ C[\bar{Y}] \triangleleft : C'[\bar{K}] \quad (\text{def } m[\bar{P}](\bar{x} : \bar{T}) : T) \in C' \\ \Delta = \{ \bar{Q} \}, \{ \bar{P} \{ (\bar{Y}') \setminus (\bar{K}) \} \} \quad \Gamma = \text{this} : C[\bar{Y}], \bar{x} : \bar{T} \{ (\bar{Y}') \setminus (\bar{K}) \} \\ \Delta; \Gamma \vdash t : S \quad \Delta \vdash S < : T \{ (\bar{Y}') \setminus (\bar{K}) \} \end{array}}{C \vdash \text{def } m = t \text{ WF}} \\
\text{(WF-CLASS-DECL)} \frac{\Delta = \{ \bar{Q} \} \quad \Delta \vdash \bar{Q} \text{ WF} \quad \Delta \vdash N_{opt} \text{ WF} \quad C \vdash \bar{M} \text{ WF}}{\text{class } C[\bar{Q}] \text{ extends } N_{opt} \{ \bar{M} \} \text{ WF}}
\end{array}$$

**Fig. 9.** Member and class well-formedness ( $C \vdash M \text{ WF}$ ,  $D \text{ WF}$ )

## 5 Syntactic Sugar

In this section we explicit the syntactic sugar that we have used in our examples and discuss some related issues. The syntactic sugar described here is of course also supported by our prototype type-checker [12].

### 5.1 Simple Syntactic Sugar

Most of the syntactic sugar is very simple. For instance, empty type parameter lists can be omitted in class and method declarations. Similarly empty type argument lists can be omitted in instance creations and method calls. The presence of a user defined class `Object` is assumed and missing superclasses and bounds in class and parameter declarations are replaced by `Object`. Finally, the special identifier `_` can be used for any parameter that is never referenced. These transformations are described by the following *rewriting rules*.

<code>class C extends ... { ... }</code>	$\longrightarrow$	<code>class C[] extends ... { ... }</code>	
<code>def m(...): ...</code>	$\longrightarrow$	<code>def m[](...): ...</code>	
<code>new C{...}</code>	$\longrightarrow$	<code>new C[] {...}</code>	
<code>e.m(...)</code>	$\longrightarrow$	<code>e.m[](...)</code>	
<code>class C[...] { ... }</code>	$\longrightarrow$	<code>class C[...] extends Object[] { ... }</code>	
<code>X[...]</code>	$\longrightarrow$	<code>X[...] &lt;: Object[]</code>	
<code>X[_[...]] &lt;: N</code>	$\longrightarrow$	<code>X[Y[...]] &lt;: N</code>	(with Y fresh)

In the monad example of Section 1.2 we have used the notation `Monad[_]` to declare a unary type constructor `Monad`. This was actually syntactic sugar for the declaration `Monad[X] <: Object`.

## 5.2 Type Constructors of Arity 0

In Section 1.2, the method `unit` was defined as follows:

```
def unit[X](x: X): Monad[X]
```

The parameter `X` looks like a type parameter but as  $\text{FGJ}_\omega$  only supports type constructor parameters, it is in fact a type constructor parameter of arity 0. Formally, it should be declared with an empty parameter list and its occurrence in the type of `x` should be followed by an empty argument list. Surprisingly, the occurrence in the return type is perfectly correct. In some sense, it is due to two counter-intuitive features of  $\text{FGJ}_\omega$  that cancel each other out; on one hand, the expression `X` does not represent a type but a type constructor and on the other hand type constructors do not take types but type constructors as arguments. Formally, the method `unit` should have been defined as follows:

```
def unit[X[]](x: X[]): Monad[X]
```

Clearly, the programmer prefers to write the original version, which is shorter. He does not want to be burdened by an isomorphism between types and type constructors of arity 0. For that reason, he would also like to be able to write `List[List[Int]]` instead of `List[[]  $\Rightarrow$  List[Int]]` i.e. to provide a type where a type constructor of arity 0 is expected.

To let the programmer use these alternative syntaxes, the syntax needs to be slightly enhanced. The syntax of parameter declarations needs to include the declarations of parameters with no explicit parameter list. The syntax of type constructors must include types and the syntax of types must include bare class names and bare parameter names. This translates into the following enhanced concrete syntax for the programmer.

$$\begin{aligned}
P &::= X[\overline{P}] <: N \mid X <: N \\
K &::= X \mid C \mid \overline{P} \Rightarrow T \mid T \\
T &::= K[\overline{K}] \mid C \mid X \\
N &::= C[\overline{K}]
\end{aligned}$$

The compiler always knows whether a type or a type constructor is expected. It is therefore able to infer that an empty argument list is missing when a type constructor occurs where a type is expected and to transform a type occurring where a type constructor is expected into an anonymous type constructor of arity 0.

With the enhanced syntax it is also possible to provide a type constructor of arity greater than zero like `List` in a context where a type is expected. Of course, in that case, the compiler will not be able to infer the missing arguments and will report an error. Similarly, it will also report an error if a type is provided where a type constructor of arity greater than zero is expected.

### 5.3 $\text{FGJ}_\omega$ as an extension of FGJ

If one disregards type casts and method overriding, which are not supported by  $\text{FGJ}_\omega$ ,  $\text{FGJ}_\omega$  and FGJ differ only in their syntax for types. By observing that the enhanced syntax of parameters and types described above is a superset of the syntax of the corresponding elements in FGJ reproduced below,  $\text{FGJ}_\omega$  can be considered as an extension of FGJ.

$$\begin{array}{ll} P ::= X <: N & \text{type parameter} \\ T ::= X \mid N & \text{type} \\ N ::= C[\overline{T}] & \text{class type} \end{array}$$

## 6 Generalized Algebraic Data Types

As another evidence of the value added to OO languages by abstract type constructors we give an original and natural interpretation of generalized algebraic data types (GADTs) [7] as an instance of the OO design pattern called Visitor [15], enhanced with abstract type constructors.

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. We speak of a *generalized* algebraic data type when an algebraic data type has a type parameter and that different cases in its definition instantiate this 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.

The classical example to illustrate GADTs is a type-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, it represents the type of the value returned by its evaluation. Thus, the case `IntLit`, for literal integers, instantiates `X` to `Int`, while the case `If`, for conditionals, instantiates `X` to the same type as its two branches (then, else). Encoding the type of the evaluation of an expression in the type of its representation enables the type-checker to reject ill-formed expressions like `new Plus(new IntLit(2), new BoolLit(false))`.

Using the visitor design pattern such a data type can be defined by the following declarations.

```
class Expr[X] {
  def matchWith[R[_]](v: Visitor[R]): R[X]
}
class Visitor[R[_]] {
```

```

    def caseIntLit(x: Int): R[Int]
    def caseBoolLit(x: Boolean): R[Boolean]
    def casePlus(x: Expr[Int], y: Expr[Int]): R[Int]
    def caseCompare(x: Expr[Int], y: Expr[Int]): R[Boolean]
    def caseIf[X](x: Expr[Boolean], y: Expr[X], z: Expr[X]): R[X]
  }
  class IntLit extends Expr[Int] {
    val x: Int
    def matchWith[R[_]](v: Visitor[R]): R[Int] =
      v.caseIntLit(this.x)
  }
  class BoolLit extends Expr[Boolean] {
    val x: Boolean
    def matchWith[R[_]](v: Visitor[R]): R[Boolean] =
      v.caseBoolLit(this.x)
  }
  class Plus extends Expr[Int] {
    val x: Expr[Int] val y: Expr[Int]
    def matchWith[R[_]](v: Visitor[R]): R[Int] =
      v.casePlus(this.x, this.y)
  }
  class Compare extends Expr[Boolean] {
    val x: Expr[Int] val y: Expr[Int]
    def matchWith[R[_]](v: Visitor[R]): R[Boolean] =
      v.caseCompare(this.x, this.y)
  }
  class If[X] extends Expr[X] {
    val x: Expr[Int] val y: Expr[X] val z: Expr[X]
    def matchWith[R[_]](v: Visitor[R]): R[X] =
      v.caseIf[X](this.x, this.y, this.z)
  }
}

```

The method `matchWith`, which applies a visitor to an expression of type `Expr[X]`, is parameterized by a unary type constructor `R` and returns an object of type `R[X]`. It means that the return type of a pattern-matching construct will functionally depend on the type `X` that characterizes the expression. The class `Visitor` is also parameterized by this type constructor `R` and declares one method for each case of the algebraic data type. Finally, there is one subclass of class `Expr` for each case of expression: `IntLit`, `BoolLit`, `Plus`, `Compare` and `If`. In these classes the implementation of the method `matchWith` just forwards its attributes to the corresponding method of the visitor argument.

A type-safe evaluator 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 function that takes an expression `e` of type `Expr[T]` and returns a value of type `T`, as shown below.

```

def eval[T](e: Expr[T]): T =
  e.matchWith[[Y] => Y](new Visitor[[Y] => Y]{
    def caseIntLit(x: Int): Int = x
    def caseBoolLit(x: Boolean): Boolean = x
  })

```

```

def casePlus(x: Expr[Int], y: Expr[Int]): Int =
  eval[Int](x) + eval[Int](y)
def caseCompare(x: Expr[Int], y: Expr[Int]): Boolean =
  eval[Int](x) < eval[Int](y)
def caseIf[X](x: Expr[Boolean], y: Expr[X], z: Expr[X]): X =
  if (eval[Boolean](x)) eval[X](y) else eval[X](z)
}

```

It works by pattern-matching the expression  $e$  using a visitor whose parameter  $R$  is instantiated with the anonymous type constructor  $[Y] \Rightarrow Y$  (the identity function at level of types). Since  $e$  is of type  $\text{Expr}[T]$  the return type of the `matchWith` method call is  $R[T]$  with  $R$  instantiated to  $[Y] \Rightarrow Y$ , which is equivalent, modulo beta-conversion, to  $T$ , the expected type for the `eval` method body. In the class `Visitor` the method `caseIntLit` is defined with the return type  $R[\text{Int}]$ . Here  $R$  is instantiated to  $[Y] \Rightarrow Y$ , thus `caseIntLit` should return a value of type `Int`, which is indeed the case. Similarly one can check that the other methods of the visitor return values of the right type.

This application of abstract type constructors is, after the one of monads, another evidence of their versatility.

## 7 Related Work

HASKELL. 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 `StateT[S]` even if `StateT` has been declared with two type parameters. Such a construct is actually a partial type application and is equivalent to  $[X] \Rightarrow \text{StateT}[S, X]$  in our design. Thus, HASKELL is able to represent all anonymous type constructors that correspond to partial type application but not the others. This is clearly a restriction compared to our design: for example, the  $\text{FGJ}_\omega$  anonymous type constructor  $[S] \Rightarrow \text{StateT}[S, X]$  is not expressible as a partial application of `StateT`. In HASKELL, which has a complete type inference mechanism based on unification, such a limitation is useful to ensure decidability of type-checking. Since we adopt in  $\text{FGJ}_\omega$  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 [16], 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. However, because of the constraint of keeping type unification decidable, they are still forced to reject some lambda-abstractions. For example, they reject the identity function at the level of types, which plays a central role in our encoding of GADTs.

SCALA *virtual types*. Some of the examples we present in this paper can be encoded in the SCALA type system using virtual types with refinements, as was proposed a long time ago by the authors on the SCALA mailing list [12], and as was recently explained in [17]. The principle is to encode a type constructor parameter declaration like `Monad[_]` as the declaration of a virtual type `Monad` bound by a class `Unary`.

```

class Unary { type Elem }
type Monad <: Unary

```

Given these definitions, the type `Monad[Int]` can be expressed by the SCALA type `Monad{type Elem = Int}`: this kind of type is called a *refined type* in SCALA, it represents all instances of `Monad` in which the type field `Elem` is equal to `Int`.

Even if this encoding is sufficient for representing monads in SCALA, it is doubtful that anonymous type constructors could be encoded in the general case.

*Reifying OO Design Patterns with Abstract Type Constructors.* Monads are a design-pattern of functional languages and they can be abstracted as a library in a language like HASKELL that has abstract type constructors. It is legitimate to ask oneself if, by adding abstract type constructors to an OO language, it could be possible to abstract over certain OO design-patterns. In [18], the author shows that abstract type constructors make it possible to implement four related design patterns (Composite, Visitor, Iterator and Builder) described in [18] as a library. For implementing this library of design patterns he uses the functional language HASKELL. This library was also successfully translated in our calculus [12], which increases our confidence in the expressiveness of our design.

*Encoding of GADTs.* In [7], it is shown that GADTs can be encoded by the visitor design pattern in any nominal OO language that allows for type equality constraints. In Section 6, we address the same problem of representing GADTs with visitors but we provide a different solution: for implementing a type-safe evaluator, we use an abstract type constructor, which is finally instantiated with an anonymous type constructor ( $[Y] \Rightarrow Y$ ). The key of both encodings is to express a *functional dependence* between the result type `R` of the visitor and the type `X` of the evaluation of an expression of type `Expr[X]`. Since type constructors are themselves functions (from types to types), a functional dependence is more naturally expressed in our formalism than with type equalities. On the other hand, there are cases where a full unification algorithm based on type equalities will be stronger than our approach.

## Conclusion

Our contributions are a compact notation for type constructor parameters in JAVA-like languages, a working prototype implementation that allowed us to typecheck all examples of the paper and a complete formalization. We also have demonstrated the expressiveness of our design by giving several powerful examples: monads, sequences polymorphic in the type of their constructor, GADTs.

There are two important properties of our calculus that would need to be proved formally, namely type soundness and decidability of typing. The type soundness proof of  $FGJ_\omega$  is likely to resemble the one of FGJ but the introduction of abstract type constructors will necessarily add some complexity in the proof since we now have to deal with types that can be reduced. Both proofs are left for future work.

Some useful features of JAVA-like languages are still absent from our calculus. Wild-card types [19, 20] have been popularized by their implementation in JAVA. A continuation of this work would be to study, at a theoretical level, the interaction between abstract type constructors and type wildcards. From a more pragmatic point of view, it would also be interesting to see if abstract type constructors are compatible with the more common strategies for inference of type arguments in JAVA-like languages.

Finally, to definitively test the usability of the design, the next step would be to implement it in a real JAVA-like language. The experience gained with our prototype type-checker will certainly be helpful for this task.

## References

1. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley)). Addison-Wesley Professional (2005)
2. Stroustrup, B.: The C++ Programming Language. Third edn. Addison Wesley Longman, Reading, MA (1997)
3. Jones, S.P.: The Haskell 98 language and libraries: The revised report. Cambridge University Press (2003)
4. 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.
5. Odersky, M., Co: The Scala Language Specification (version 2.0). <http://scala.epfl.ch/docu/files/ScalaReference.pdf> (November 2006)
6. 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.
7. Kennedy, A., Russo, C.: Generalized algebraic data types and object-oriented programming. In: the 2005 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), ACM Press (October 2005)
8. Cook, W.R.: Object-oriented programming versus abstract data types. In: Proceedings of the REX School/Workshop on Foundations of Object-Oriented Languages, London, UK, Springer-Verlag (1991) 151–178
9. Canning, P., Cook, W., Hill, W., Olthoff, W., Mitchell, J.C.: F-bounded polymorphism for object-oriented programming. In: FPCA '89: Proceedings of the fourth international conference on Functional programming languages and computer architecture, New York, NY, USA, ACM Press (1989) 273–280
10. de Bruijn, N.G.: Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.* **34**(5) (1972) 381–392
11. Project, T.C.D.T.L.: The Coq proof assistant reference manual (version 8.0). <http://coq.inria.fr> (2004)
12. Altherr, P., Cremet, V.: Abstract type constructors for Java-like languages (web page). <http://lamp.epfl.ch/~cremet/fgjomega> (December 2006)
13. Barendregt, H.P.: The Lambda Calculus – Its Syntax and Semantics. Revised edn. North Holland (1984)
14. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, Massachusetts (1994)
16. Neubauer, M., Thiemann, P.: Type classes with more higher-order polymorphism. In: ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming, New York, NY, USA, ACM Press (2002) 179–190
17. Moors, A., Piessens, F., Joosen, W.: An object-oriented approach to datatype-generic programming. In: Workshop on Generic Programming (WGP'2006), ACM (September 2006)
18. Gibbons, J.: Design patterns as higher-order datatype-generic programs. In: Workshop on Generic Programming (WGP'2006), ACM (September 2006)

19. Igarashi, A., Viroli, M.: On variance-based subtyping for parametric types (June 2002)
20. 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