Generics of a Higher Kind

Adriaan Moors Frank Piessens
DistriNet, K.U.Leuven
{adriaan, frank}@cs.kuleuven.be

Martin Odersky
EPFL
martin.odersky@epfl.ch

Abstract

With Java 5 and C# 2.0, first-order parametric polymorphism was introduced in mainstream object-oriented programming languages under the name of *generics*. Although the first-order variant of generics is very useful, it also imposes some restrictions: it is possible to abstract over a type, but the resulting type constructor cannot be abstracted over. This can lead to code duplication. We removed this restriction in Scala, by allowing type constructors as type parameters and abstract type members. This paper presents the design and implementation of the resulting type constructor polymorphism. Furthermore, we study how this feature interacts with existing object-oriented constructs, and show how it makes the language more expressive.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Polymorphism

General Terms Design, Experimentation, Languages

Keywords type constructor polymorphism, higher-kinded types, higher-order genericity, Scala

1. Introduction

First-order parametric polymorphism is now a standard feature of statically typed programming languages. Starting with System F [23, 50] and functional programming languages, the constructs have found their way into object-oriented languages such as Java, C#, and many more. In these languages, first-order parametric polymorphism is usually called *generics*. Generics rest on sound theoretical foundations, which were established by Abadi and Cardelli [2, 1], Igarashi et al. [31], and many others; they are well-understood by now.

One standard application area of generics are collections. For instance, the type `List[A]` represents lists of a given element type `A`, which can be chosen freely. In fact, generics can be seen as a generalisation of the type of arrays, which has always been parametric in the type of its elements.

First-order parametric polymorphism has some limitations, however. Although it allows to abstract over types, which yields *type constructors* such as `List`, these type constructors cannot be abstracted over. For instance, one cannot pass a type constructor as a type argument to another type constructor. Abstractions that require this are quite common, even in object-oriented programming, and this restriction thus leads to unnecessary duplication of code. We provide several examples of such abstractions in this paper.

The generalisation of first-order polymorphism to a higher-order system was a natural step in lambda calculus [23, 50, 7]. This theoretical advance has since been incorporated into functional programming languages. For instance, the Haskell programming language [28] supports type constructor polymorphism, which is also integrated with its type class concept [33]. This generalisation to types that abstract over types that abstract over types ("higher-kinded types") has many practical applications. For example, comprehensions [54], parser combinators [30, 35], as well as more recent work on embedded Domain Specific Languages (DSL's) [14, 26] critically rely on higher-kinded types.

The same needs – as well as more specific ones – arise in object-oriented programming. LINQ brought direct support for comprehensions to the .NET platform [5, 37], Scala [43] has had a similar feature from the start, and Java 5 introduced a lightweight variation [24, Sec. 14.14.2]. Parser combinators are also gaining momentum: Bracha uses them as the underlying technology for his Executable Grammars [6], and Scala's distribution includes a library [39] that implements an embedded DSL for parsing, which allows users to express parsers directly in Scala, in a notation that closely resembles EBNF. Type constructor polymorphism is crucial in defining a common parser interface that is implemented by different back-ends.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.

In this paper, we focus on our experience with extending Scala with type constructor polymorphism, and on the resulting gain in expressivity of the language as a whole. A similar extension could be added to, for example, Java in the same way [3]. Our extension was incorporated in Scala 2.5, which was released in May 2007.

The main contributions of this paper are as follows:

- We illustrate the utility and practicality of type constructor polymorphism using a realistic example.
- We develop a kind system that captures both lower and upper bounds, and variances of types.
- We survey how the integration with existing features of Scala (such as subtyping, definition-site variance annotations, and implicit arguments) makes the language more powerful.
- We relate our experience with implementing the kind system in the open-source Scala compiler.

For the reader who is not yet familiar with Scala, the next section provides a brief introduction. The rest of this paper is divided in three parts, which each consider a different facet of the evaluation of type constructor polymorphism. First, Section 3 demonstrates that our extension reduces boilerplate that arises from the use of genericity. We establish intuitions with a simple example, and extend it to a realistic implementation of the comprehensions fragment of `Iterable`.

Second, we present the type and kind system. Section 4 discusses the surface syntax in full Scala, and the underlying model of kinds. Based on the ideas established in the theoretical part, Section 5 refines `Iterable`, so that it accommodates collections that impose bounds on the type of their elements.

Third, we have validated the practicality of our design by implementing our extension in the Scala compiler, and we report on our experience in Section 6. Throughout the paper, we discuss various interactions of type constructor polymorphism with existing features in Scala. Section 7 focusses on the integration with Scala's implicits, which are used to encode Haskell's type classes. Our extension lifts this encoding to type *constructor* classes. Furthermore, due to subtyping, Scala supports abstracting over type class contexts, so that the concept of a bounded monad can be expressed cleanly, which is not possible in (mainstream extensions of) Haskell.

Finally, we summarise related work in Section 8 and conclude in Section 9.

2. Prelude: Scala Basics

This section introduces the basic subset of Scala [43, 45] that is used in the examples of this paper. We assume familiarity with a Java-like language, and focus on what makes Scala different.

2.1 Outline of the syntax

A Scala program is roughly structured as a tree of nested definitions. A definition starts with a keyword, followed by its name, a classifier, and the entity to which the given name is bound, if it is a concrete definition. If the root of the tree is the compilation unit, the next level consists of objects (introduced by the keyword **`object`**) and classes (**`class`**, **`trait`**), which in turn contain members. A member may again be a class or an object, a constant value member (**`val`**), a mutable value member (**`var`**), a method (**`def`**), or a type member (**`type`**). Note that a type annotation always follows the name (or, more generally, the expression) that it classifies.

On the one hand, Scala's syntax is very regular, with the *keyword/name/classifier/bound entity*-sequence being its lead motif. Another important aspect of this regularity is nesting, which is virtually unconstrained. On the other hand, syntactic sugar enables flexibility and succinctness. For example, `buffer += 10` is shorthand for the method call `buffer.+=(10)`, where `+=` is a user-definable identifier.

2.2 Functions

Since Scala is a functional language, functions are first-class values. Thus, like an integer, a function can be written down directly: `x: Int ⇒ x + 1` is the successor function on integers. Furthermore, a function can be passed as an argument to a (higher-order) function or method. Functions and methods are treated similarly in Scala, the main difference is that a method is called on a target object.

The following definition introduces a function `len` that takes a `String` and yields an `Int` by calling `String`'s `length` method on its argument `s`:

```
val len: String ⇒ Int = s ⇒ s.length
```

In the classifier of the definition, the type `String ⇒ Int`, the arrow ⇒ is a type constructor, whereas it introduces an anonymous function on the right-hand side (where a value is expected). This anonymous function takes an argument `s` of type `String` and returns `s.length`. Thus, the application `len("four")` yields `4`.

Note that the Scala compiler infers [46] the type of the argument `s`, based on the expected type of the value `len`. The direction of type inference can also be reversed:

```
val len = (s: String) ⇒ s.length
```

The right-hand side's anonymous function can be abbreviated using syntactic sugar that implicitly introduces functional abstraction. This can be thought of as turning `String`'s `length` method into a function:

```
val len: String ⇒ Int = _.length
```

Finally, since Scala is purely object-oriented at its core, a function is represented internally as an object with an `apply`

method that is derived straightforwardly from the function. Thus, one more equivalent definition of `len`:

```
object len {
  def apply(s: String): Int = s.length
}
```

2.3 Classes, traits, and objects

In Scala, a class can inherit from another class and one or more traits. A trait is a class that can be composed with other traits using mixin composition. Mixin composition is a restricted form of multiple inheritance, which avoids ambiguities by linearising the graph that results from composing classes that are themselves composites. The details are not relevant for this paper, and we simply refer to both classes and traits as "classes".

The feature that is relevant to this paper, is that classes may contain *type* members. An abstract type member is similar to a type parameter. The main difference between parameters and members is their scope and visibility. A type parameter is syntactically part of the type that it parameterises, whereas a type member – like value members – is encapsulated, and must be selected explicitly. Similarly, type members are inherited, while type parameters are local to their class. The complementary strengths of type parameters and abstract type members are a key ingredient of Scala's recipe for scalable component abstractions [47].

Type parameters are made concrete using type application. Thus, given the definition **`class`** `List[T]`, `List` is a type constructor (or type function), and `List[Int]` is the application of this function to the argument `Int`. Abstract type members are made concrete using *abstract type member refinement*, a special form of mixin composition. Note that `List` is now an abstract class[1], since it has an abstract member `T`:

```
trait List {
  type T
}
```

This abstract member is made concrete as follows:

```
List{type T=Int}
```

Note that, with our extension, type members may also be parameterised, as in **`type`** `Container[X]`.

Methods typically define one or more lists of value parameters, in addition to a list of type parameters. Thus, a method can be seen as a value that abstracts over values and types. For example, **`def`** `iterate[T](a: T)(next: T ⇒ T, done: T ⇒ Boolean): List[T]` introduces a method with one type parameter `T`, and two argument lists. Methods with multiple argument lists may be partially applied. For example, for some object `x` on which `iterate` is defined, `x.iterate(0)` corresponds to a higher-order function with type `(Int ⇒ Int, Int ⇒ Boolean) ⇒ List[Int]`. Note that the type parameter `T` was inferred to be `Int` from the type of `a`.

Finally, an **`object`** introduces a class with a singleton instance, which can be referred to using the object's name.

3. Reducing Code Duplication with Type Constructor Polymorphism

This section illustrates the benefits of generalising genericity to type constructor polymorphism using the well-known `Iterable` abstraction. The first example, which is due to Lex Spoon, illustrates the essence of the problem in the small. Section 3.1 extends it to more realistic proportions.

Listing 1 shows a Scala implementation of the trait `Iterable[T]`. It contains an abstract method `filter` and a convenience method `remove`. Subclasses should implement `filter` so that it creates a new collection by retaining only the elements of the current collection that satisfy the predicate `p`. This predicate is modelled as a function that takes an element of the collection, which has type `T`, and returns a `Boolean`. As `remove` simply inverts the meaning of the predicate, it is implemented in terms of `filter`.

Naturally, when filtering a list, one expects to again receive a list. Thus, `List` overrides `filter` to refine its result type covariantly. For brevity, `List`'s subclasses, which implement this method, are omitted. For consistency, `remove` should have the same result type, but the only way to achieve this is by overriding it as well. The resulting code duplication is a clear indicator of a limitation of the type system: both methods in `List` are redundant, but the type system is not powerful enough to express them at the required level of abstraction in `Iterable`.

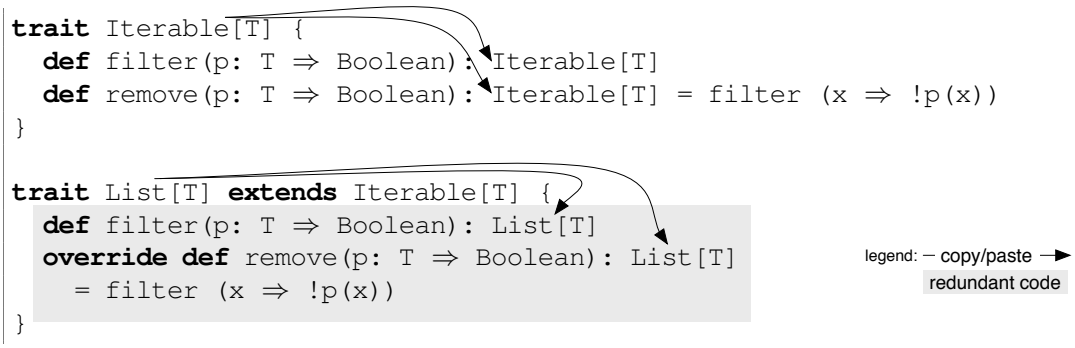
Our solution, depicted in Listing 2, is to abstract over the type constructor that represents the container of the result of `filter` and `remove`. The improved `Iterable` now takes two type parameters: the first one, `T`, stands for the type of its elements, and the second one, `Container`, represents the *type constructor* that determines part of the result type of the `filter` and `remove` methods. More specifically, `Container` is a type parameter that itself takes one type parameter. Although the name of this higher-order type parameter (`X`) is not needed here, more sophisticated examples will show the benefit of explicitly naming[2] higher-order type parameters.

Now, to denote that applying `filter` or `remove` to a `List[T]` returns a `List[T]`, `List` simply instantiates `Iterable`'s type parameter to the `List` type constructor.

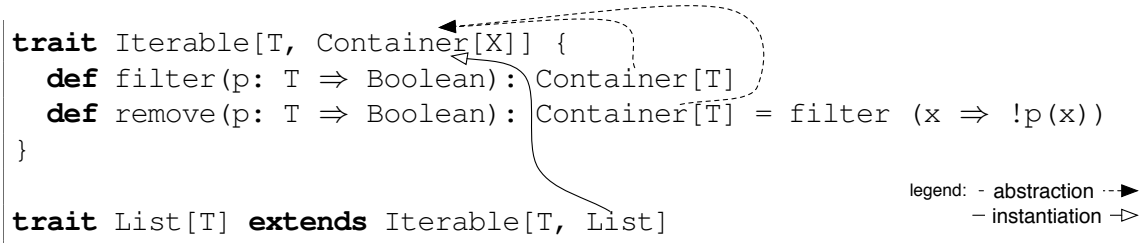
In this simple example, one could also use a construct like Bruce's `MyType` [9]. However, this scheme breaks down in more complex cases, as demonstrated in the next section.

[1] For brevity, we use the **`trait`** keyword instead of **`abstract class`**.

[2] In full Scala '_' may be used as a wild-card name for higher-order type parameters.



Listing 1. Limitations of Genericity



Listing 2. Removing Code Duplication

3.1 Improving Iterable

In this section we design and implement the abstraction that underlies comprehensions [54]. Type constructor polymorphism plays an essential role in expressing the design constraints, as well as in factoring out boilerplate code without losing type safety. More specifically, we discuss the signature and implementation of `Iterable`'s `map`, `filter`, and `flatMap` methods. The LINQ project brought these to the .NET platform as `Select`, `Where`, and `SelectMany` [36].

Comprehensions provide a simple mechanism for dealing with collections by transforming their elements (`map`, `Select`), retrieving a sub-collection (`filter`, `Where`), and collecting the elements from a collection of collections in a single collection (`flatMap`, `SelectMany`).

To achieve this, each of these methods interprets a user-supplied function in a different way in order to derive a new collection from the elements of an existing one: `map` transforms the elements as specified by that function, `filter` interprets the function as a predicate and retains only the elements that satisfy it, and `flatMap` uses the given function to produce a collection of elements for every element in the original collection, and then collects the elements in these collections in the resulting collection.

The only collection-specific operations that are required by a method such as `map`, are iterating over a collection, and producing a new one. Thus, if these operations can be abstracted over, these methods can be implemented in `Iterable` in terms of these abstractions. Listing 3 shows the well-known, lightweight, `Iterator` abstraction that encapsulates iterating over a collection, as well as the `Builder` abstraction, which captures how to produce a collection, and thus may be thought of as the dual of `Iterator`.

```
trait Builder[Container[X], T] {
  def +=(el: T): Unit
  def finalise(): Container[T]
}

trait Iterator[T] {
  def next(): T
  def hasNext: Boolean

  def foreach(op: T ⇒ Unit): Unit
    = while(hasNext) op(next())
}
```

Listing 3. `Builder` and `Iterator`

`Builder` crucially relies on type constructor polymorphism, as it must abstract over the type constructor that represents the collection that it builds. The `+=` method is used to supply the elements in the order in which they should appear in the collection. The collection itself is returned by `finalise`. For example, the `finalise` method of a `Builder[List, Int]` returns a `List[Int]`.

Listing 4 shows a minimal `Buildable` with an abstract `build` method, and a convenience method, `buildWith`, that captures the typical use-case for `build`.

By analogy to the proven design that keeps `Iterator` and `Iterable` separated, `Builder` and `Buildable` are modelled as separate abstractions as well. In a full implementation, `Buildable` would contain several more methods, such as `unfold` (the dual of `fold` [22]), which should not clutter the lightweight `Builder` interface.

Note that `Iterable` uses a type constructor member, `Container`, to abstract over the precise type of the container, whereas `Buildable` uses a parameter. Since clients of `Iterable` generally are not concerned with the exact type of the container (except for the regularity that is imposed by our design), it is neatly encapsulated as a type member. `Buildable`'s primary purpose is exactly to create and populate a specific kind of container. Thus, the type of an instance of the `Buildable` class should specify the type of container that it builds. This information is still available with a type member, but it is less manifest.

The `map`/`filter`/`flatMap` methods are implemented in terms of the even more flexible trio `mapTo`/`filterTo`/`flatMapTo`. The generalisation consists of decoupling the original collection from the produced one – they need not be the same, as long as there is a way of building the target collection. Thus, these methods take an extra argument of type `Buildable[C]`. Section 7 shows how an orthogonal feature of Scala can be used to relieve callers from supplying this argument explicitly.

For simplicity, the `mapTo` method is implemented as straightforwardly as possible. The `filterTo` method shows how the `buildWith` convenience method can be used.

The result types of `map`, `flatMap`, and their generalisations illustrate why a `MyType`-based solution would not work: whereas the type of **`this`** would be `C[T]`, the result type of these methods is `C[U]`: it is the same type *constructor*, but it is applied to different type *arguments*!

Listings 5 and 6 show the objects that implement the `Buildable` interface for `List` and `Option`. An `Option` corresponds to a list that contains either 0 or 1 elements, and is commonly used in Scala to avoid `null`'s.

With all this in place, `List` can easily be implemented as a subclass of `Iterable`, as shown in Listing 7. The type constructor of the container is fixed to be `List` itself, and the standard `Iterator` trait is implemented. This implementation does not offer any new insights, so we have omitted it.

3.2 Example: using Iterable

This example demonstrates how to use `map` and `flatMap` to compute the average age of the users of, say, a social networking site. Since users do not have to enter their birthday, the input is a `List[Option[Date]]`. An `Option[Date]`

```
trait Buildable[Container[X]] {
  def build[T]: Builder[Container, T]

  def buildWith[T](f: Builder[Container,T]⇒
                      Unit): Container[T] ={
    val buff = build[T]
    f(buff)
    buff.finalise()
  }
}

trait Iterable[T] {
  type Container[X] <: Iterable[X]

  def elements: Iterator[T]

  def mapTo[U, C[X]](f: T ⇒ U)
           (b: Buildable[C]): C[U] = {
    val buff = b.build[U]
    val elems = elements

    while(elems.hasNext){
      buff += f(elems.next)
    }
    buff.finalise()
  }
  def filterTo[C[X]](p: T ⇒ Boolean)
             (b: Buildable[C]): C[T] = {
    val elems = elements

    b.buildWith[T]{ buff ⇒
      while(elems.hasNext){
        val el = elems.next
        if(p(el)) buff += el
      }
    }
  }
  def flatMapTo[U,C[X]](f: T⇒Iterable[U])
              (b: Buildable[C]): C[U] = {
    val buff = b.build[U]
    val elems = elements

    while(elems.hasNext){
      f(elems.next).elements.foreach{ el ⇒
        buff += el
      }
    }
    buff.finalise()
  }

  def map[U](f: T ⇒ U)
      (b: Buildable[Container]): Container[U]
    = mapTo[U, Container](f)(b)
  def filter(p: T ⇒ Boolean)
      (b: Buildable[Container]): Container[T]
    = filterTo[Container](p)(b)
  def flatMap[U](f: T ⇒ Container[U])
      (b: Buildable[Container]): Container[U]
    = flatMapTo[U, Container](f)(b)
}
```

Listing 4. `Buildable` and `Iterable`

```
object ListBuildable extends Buildable[List]{
  def build[T]: Builder[List, T] = new
    ListBuffer[T] with Builder[List, T] {
    // += is inherited from ListBuffer (Scala
     standard library)
    def finalise(): List[T] = toList
  }
}
```

Listing 5. Building a `List`

```
object OptionBuildable extends
                        Buildable[Option] {
def build[T]: Builder[Option, T]
  = new Builder[Option, T] {
      var res: Option[T] = None()

      def +=(el: T)
        = if(res.isEmpty) res = Some(el)
          else throw new UnsupportedOperation
    -Exception(">1␣elements")

      def finalise(): Option[T] = res
    }
}
```

Listing 6. Building an `Option`

```
class List[T] extends Iterable[T]{
  type Container[X] = List[X]

  def elements: Iterator[T]
    = new Iterator[T] {
        // standard implementation
    }
}
```

Listing 7. `List` subclasses `Iterable`

either holds a date or nothing. Listing 8 shows how to proceed.

First, a small helper is introduced that computes the current age in years from a date of birth. To collect the known ages, an optional date is transformed into an optional age using `map`. Then, the results are collected into a list using `flatMapTo`. Note the use of the more general `flatMapTo`. With `flatMap`, the inner `map` would have had to convert its result from an `Option` to a `List`, as `flatMap(f)` returns its results in the same kind of container as produced by the function `f` (the inner `map`). Finally, the results are aggregated using `reduceLeft` (not shown here). The full code of the example is available on the paper's homepage[3].

Note that the Scala compiler infers most proper types (we added some annotations to aid understanding), but it does not

[3] http://www.cs.kuleuven.be/~adriaan/?q=genericshk

```
val bdays: List[Option[Date]] = List(
  Some(new Date("1981/08/07")), None,
  Some(new Date("1990/04/10")))
def toYrs(bd: Date): Int = // omitted

val ages: List[Int]
 = bdays.flatMapTo[Int, List]{ optBd ⇒
    optBd.map{d ⇒ toYrs(d)}(OptionBuildable)
   }(ListBuildable)

val avgAge = ages.reduceLeft[Int](_ + _) /
                ages.length
```

Listing 8. Example: using `Iterable`

infer type constructor arguments. Thus, type argument lists that contain type constructors, must be supplied manually.

Finally, the only type constructor that arises in the example is the `List` type argument, as type constructor inference has not been implemented yet. This demonstrates that the complexity of type constructor polymorphism, much like with genericity, is concentrated in the internals of the library. The upside is that library designers and implementers have more control over the interfaces of the library, while clients remain blissfully ignorant of the underlying complexity. (As noted earlier, Section 7 will show how the arguments of type `Buildable[C]` can be omitted.)

3.3 Members versus parameters

The relative merits of abstract members and parameters have been discussed in detail by many others [8, 53, 21]. The Scala philosophy is to embrace both: sometimes parameterisation is the right tool, and at other times, abstract members provide a better solution. Technically, it has been shown how to safely encode parameters as members [40], which – surprisingly – wasn't possible in earlier calculi [44].

Our examples have used both styles of abstraction. `Buildable`'s main purpose is to build a certain container. Thus, `Container` is a type parameter: a characteristic that is manifest to external clients of `Buildable`, as it is (syntactically) part of the type of its values. In `Iterable` a type member is used, as its external clients are generally only interested in the type of its elements. Syntactically, type members are less visible, as `Iterable[T]` is a valid proper type. To make the type member explicit, one may write `Iterable[T]{type Container[X]=List[X]}`. Alternatively, the `Container` type member can be selected on a singleton type that is a subtype of `Iterable[T]`.

4. Of Types and Kinds

Even though proper types and type constructors are placed on equal footing as far as parametric polymorphism is concerned, one must be careful not to mix them up. Clearly, a type parameter that stands for a proper type, must not be

```
TypeParamClause  ::= '[' TypeParam {',' TypeParam} ']'
TypeParam        ::=  id [TypeParamClause] ['>:' Type] ['<:' Type]

AbstractTpMem    ::=  'type' TypeParam
```

Figure 1. Syntax for type declarations (type parameters and abstract type members)

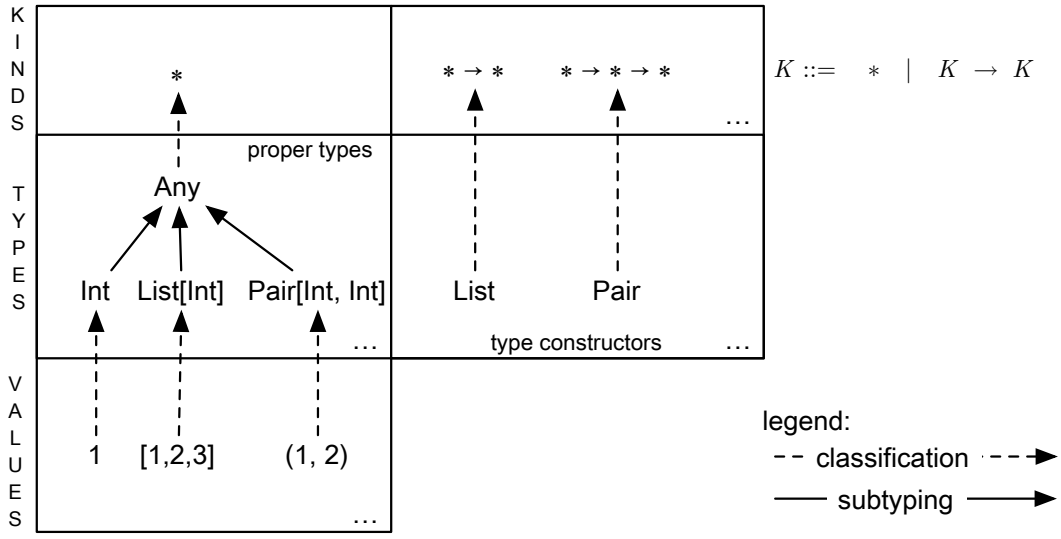


Figure 2. Diagram of levels

applied to type arguments, whereas a type constructor parameter cannot classify a value until it has been turned into a proper type by supplying the right type arguments.

In this section we give an informal overview of how programmers may introduce higher-kinded type parameters and abstract type members, and sketch the rules that govern their use. We describe the surface syntax that was introduced with the release of Scala 2.5, and the underlying conceptual model of kinds.

4.1 Surface syntax for types

Figure 1 shows a simplified fragment of the syntax of type parameters and abstract type members, which we collectively call "type declarations". The full syntax, which additionally includes variance annotations, is described in the Scala language specification [42]. Syntactically, our extension introduces an optional `TypeParamClause` as part of a type declaration. The scope of the higher-order type parameters that may thus be introduced, extends over the outer type declaration to which they belong.

For example, `Container[X]` is a valid `TypeParam`, which introduces a type constructor parameter that expects one type argument. To illustrate the scoping of higher-order type parameters, `Container[X] <: Iterable[X]` declares a type parameter that, when applied to a type argument `Y` – written as `Container[Y]` – must be a subtype of `Iterable[Y]`.

As a more complicated example, `C[X <: Ordered[X]] <: Iterable[X]` introduces a type constructor parameter `C`, with an F-bounded higher-order type parameter `X`, which occurs in its own bound as well as in the bound of the type parameter that it parameterises. Thus, `C` abstracts over a type constructor so that, for any `Y` that is a subtype of `Ordered[Y]`, `C[Y]` is a subtype of `Iterable[Y]`

4.2 Kinds

Conceptually, kinds are used to distinguish a type parameter that stands for a proper type, such as `List[Int]`, from a type parameter that abstracts over a type constructor, such as `List`. An initial, simplistic kind system is illustrated in the diagram in Fig. 2, and it is refined in the remainder of this section. The figure shows the three levels of classification, where entities in lower levels are classified by entities in the layer immediately above them.

Kinds populate the top layer. The kind $\star$ classifies types that classify values, and the $\to$ kind constructor is used to construct kinds that classify type constructors. Note that kinds are inferred by the compiler. They cannot appear in Scala's surface syntax.

Nonetheless, Fig. 3 introduces syntax for the kinds that classify the types that can be declared as described in the previous section. The first kind, $\star$`(T, U)`, classifies proper types (such as type declarations without higher-order type parameters), and tracks their lower (`T`) and upper bounds

```
Kind ::= '*(' Type ',' Type ')'
       | [id '@' ] Kind '->' Kind
```

Figure 3. Kinds (not in surface syntax)

(`U`). It should be clear that this kind is easily inferred, as type declarations either explicitly specify bounds or receive the minimal lower bound, `Nothing`, and the maximal upper bound, `Any`. Note that intersection types can be used to specify a disjunction of lower bounds, and a conjunction of upper bounds. Since we mostly use upper bounds, we abbreviate `*(Nothing, T)` to `*(T)`, and `*(Nothing, Any)` is written as `*`.

We refine the kind of type constructors by turning it into a *dependent* function kind, as higher-order type parameters may appear in their own bounds, or in the bounds of their outer type parameter.

In the examples that was introduced above, `Container[X]` introduces a type constructor parameter of kind `* → *`, and `Container[X] <: Iterable[X]` implies the kind `X @ * → *(Iterable[X])` for `Container`. Finally, the declaration `C[X <: Ordered[X]] <: Iterable[X]` results in `C` receiving the kind `X @ *(Ordered[X]) → *(Iterable[X])`. Again, the syntax for higher-order type parameters provides all the necessary information to infer a (dependent) function kind for type constructor declarations.

Informally, type constructor polymorphism introduces an indirection through the kinding rules in the typing rule for type application, so that it uniformly applies to generic classes, type constructor parameters, and abstract type constructor members. These type constructors, whether concrete or abstract, are assigned function kinds by the kind system. Thus, if `T` has kind `X @ K → K'`, and `U` has kind `K`, in which `X` has been replaced by `U`, a type application `T[U]` has kind `K'`, with the same substitution applied. Multiple type arguments are supported through the obvious generalisation (taking the necessary care to perform simultaneous substitutions).

4.3 Subkinding

Similar to the subtyping relation that is defined on types, subkinding relates kinds. Thus, we overload `<:` to operate on kinds as well as on types. As the bounds-tracking kind stems from Scala's bounds on type declarations, subkinding for this kind simply follows the rules that were already defined for type member conformance: `*(T, U) <: *(T', U')` if `T' <: T` and `U <: U'`. Intuitively, this amounts to interval inclusion. For the dependent function kind, we transpose subtyping of dependent function types [4] to the kind level.

```
class Iterable[Container[X], T]
trait NumericList[T <: Number] extends
    Iterable[NumericList, T]
```

Listing 9. `NumericList`: an illegal subclass of `Iterable`

```
class Iterable[Container[X <: Bound], T <:
    Bound, Bound]

trait NumericList[T <: Number] extends
    Iterable[NumericList, T, Number]
```

Listing 10. Safely subclassing `Iterable`

4.4 Example: why kinds track bounds

Suppose `Iterable`[4] is subclassed as in Listing 9. This program is rejected by the compiler because the type application `Iterable[NumericList, T]` is ill-kinded. The kinding rules classify `NumericList` as a `*(Number) → *`, which must be a subkind of the expected kind of `Iterable`'s first type parameter, `* → *`. Now, `*(Number) <: *`, whereas subkinding for function kinds requires the argument kinds to vary contravariantly.

Intuitively, this type application must be ruled out, because passing `NumericList` as the first type argument to `Iterable` would "forget" that `NumericList` may only contain `Number`'s: `Iterable` is kind-checked under the assumption that its first type argument does not impose any bounds on its higher-order type parameter, and it could thus apply `NumericList` to, say, `String`. The next section elaborates on this.

Fortunately, `Iterable` can be defined so that it can accommodate bounded collections, as shown in Listing 10. To achieve this, `Iterable` abstracts over the bound on `Container`'s type parameter. `NumericList` instantiates this bound to `Number`. We refine this example in Section 5.

4.5 Kind soundness

Analogous to type soundness, which provides guarantees about value-level abstractions, kind soundness ensures that type-level abstractions do not go "wrong".

At the value level, passing, e.g., a `String` to a function that expects an `Integer` *goes wrong* when that function invokes an `Integer`-specific operation on that `String`. Type soundness ensures that application is type-preserving, in the sense that a well-typed application evaluates to a well-typed result.

As a type-level example, consider what happens when a type function that expects a type of kind `* → *`, is applied to a type of kind `*(Number) → *`. This application *goes*

[4] For simplicity, we define `Iterable` using type parameters in this example.

```
trait Builder[Container[X <: B[X]], T <: B[T],
    B[Y]]
trait Buildable[Container[X <: B[X]], B[Y]] {
  def build[T <: B[T]]: Builder[Container,T,B]
}
trait Iterable[T <: Bound[T], Bound[X]] {
  type Container[X <: Bound[X]] <: Iterable[X,
    Bound]

  def map[U <: Bound[U]](f: T ⇒ U)
    (b: Buildable[Container, Bound]):
    Container[U] = ...
}
```

Listing 11. Essential changes to extend Iterable with support for (F-)bounds

```
class List[T] extends Iterable[T, Any] {
  type Container[X] = List[X]
}

trait OrderedCollection[T <: Ordered[T]]
    extends Iterable[T, Ordered] {
  type Container[X <: Ordered[X]] <:
    OrderedCollection[X]
}

trait Wrap1[T]{type Apply[X]=T}

trait Number
class NumericList[T <: Number] extends
    Iterable[T, Wrap1[Number]#Apply] {
  type Container[X <: Number] = NumericList[X]
}
```

Listing 12. (Bounded) subclasses of Iterable

wrong, even though the type function itself is well-kinded, if it does something with that type constructor that would be admissible with a type of kind $\star \rightarrow \star$, but not with a type of kind $\star$(Number) $\rightarrow \star$, such as applying it to String. If the first, erroneous, type application were considered well-kinded, type application would not be kind-preserving, as it would turn a well-kinded type into a nonsensical, ill-kinded, one (such as NumericList[String]).

As our kind system is closely related to dependently typed lambda calculus with subtyping, it is reasonable to assume that it is sound. Proving this conjecture, as well as the more familiar meta-theoretic results, is ongoing work. The underlying theory – an object-oriented calculus – has been described in earlier work [40].

Finally, it is important to note that kind unsoundness results in type applications "going wrong" *at compile time*. Thus, the problem is less severe than with type unsoundness, but these errors can be detected earlier in the development process, without effort from the programmer.

5. Bounded Iterable

As motivated in Section 4.4, in order for Iterable to model collections that impose an (F-)bound on the type of their elements, it must accommodate this bound from the start.

To allow subclasses of Iterable to declare an (F-)bound on the type of their elements, Iterable must abstract over this bound. Listing 11 generalises the interface of the original Iterable from Listing 4. The implementation is not affected by this change.

Listing 12 illustrates various kinds of subclasses, including List, which does not impose a bound on the type of its elements, and thus uses Any as its bound (Any and Nothing are kind-overloaded). Note that NumericList can also be derived, by encoding the anonymous type function X → Number as Wrap1[Number]#Apply.

Again, the client of the collections API is not exposed to the relative complexity of Listing 11. However, without it, a significant fraction of the collection classes could not be unified under the same Iterable abstraction. Thus, the clients of the library benefit, as a unified interface for collections, whether they constrain the type of their elements or not, means that they need to learn fewer concepts.

Alternatively, it would be interesting to introduce kind-level abstraction to solve this problem. Tentatively, Iterable and List could then be expressed as:

```
trait Iterable[T : ElemK, ElemK : Kind]
class List[T] extends Iterable[T, *]
```

This approach is more expressive than simply abstracting over the upper bound on the element type, as the interval kind can express lower and upper bounds simultaneously. This would become even more appealing in a language that allows the user to define new kinds [51].

6. Full Scala

In this section we discuss our experience with extending the full Scala compiler with type constructor polymorphism. As discussed below, the impact[5] of our extension is mostly restricted to the type checker. Finally, we list the limitations of our implementation, and discuss the interaction with variance. The implementation supports variance annotations on higher-order type parameters, but this has not been integrated in the formalisation yet.

6.1 Implementation

Extending the Scala compiler with support for type constructor polymorphism came down to introducing another level of indirection in the well-formedness checks for types.

Once abstract types could be parameterised (a simple extension to the parser and the abstract syntax trees), the

[5] The initial patch to the compiler can be viewed at http://lampsvn.epfl.ch/trac/scala/changeset/10642

check that type parameters must always be proper types had to be relaxed. Instead, a more sophisticated mechanism tracks the kinds that are inferred for these abstract types. Type application then checks two things: the type that is used as a type constructor must indeed have a function kind, and the kinds of the supplied arguments must conform to the expected kinds. Additionally, one must ensure that type constructors do not occur as the type of a value.

Since Scala uses type erasure in the back-end, the extent of the changes is limited to the type checker. Clearly, our extension thus does not have any impact on the run-time characteristics of a program. Ironically, as type erasure is at the root of other limitations in Scala, it was an important benefit in implementing type constructor polymorphism.

Similar extensions in languages that target the .NET platform face a tougher challenge, as the virtual machine has a richer notion of types and thus enforces stricter invariants. Unfortunately, the model of types does not include higher-kinded types. Thus, to ensure full interoperability with genericity in other languages on this platform, compilers for languages with type constructor polymorphism must resort to partial erasure, as well as code specialisation in order to construct the necessary representations of types that result from abstract type constructors being applied to arguments.

6.1.1 Limitations

Syntactically, there are a few limitations that we would like to lift in upcoming versions. As it stands, we do not directly support partial type application and currying, or anonymous type functions. However, these features can be encoded, as illustrated in Section 5.

We have not yet extended the type inferencer to infer higher-kinded types. In all likelihood, type constructor inference will have to be limited to a small subset in order to ensure decidability.

6.2 Variance

Another facet of the interaction between subtyping and type constructors is seen in Scala's support for definition-site variance annotations [19]. Variance annotations provide the information required to decide subtyping of types that result from applying the same type constructor to different types.

As the classical example, consider the definition of the class of immutable lists, `class List[+T]`. The `+` before `List`'s type parameter denotes that `List[T]` is a subtype of `List[U]` if `T` is a subtype of `U`. We say that `+` introduces a covariant type parameter, `-` denotes contravariance (the subtyping relation between the type arguments is the inverse of the resulting relation between the constructed types), and the lack of an annotation means that these type arguments must be identical.

Variance annotations pose the same kind of challenge to the model of kinds as did bounded type parameters: kinds must encompass them as they represent information that should not be glossed over when passing around type constructors. The same strategy as for including bounds into $\star$ can be applied here, except that variance is a property of type *constructors*, so it should be tracked in $\rightarrow$, by distinguishing $\xrightarrow{+}$ and $\xrightarrow{-}$ [52].

Without going in too much detail, we illustrate the need for variance annotations on higher-order type parameters and how they influence kind conformance.

Listing 13 defines a perfectly valid `Seq` abstraction, albeit with a contrived `lift` method. Because `Seq` declares `C`'s type parameter `X` to be covariant, it may use its covariant type parameter `A` as an argument for `C`, so that `C[A] <: C[B]` when `A <: B`.

`Seq` declares the type of its **`this`** variable to be `C[A]` (`self: C[A] ⇒` declares `self` as an alias for **`this`**, and gives it an explicit type). Thus, the `lift` method may return **`this`**, as its type can be subsumed to `C[B]`.

Suppose that a type constructor that is invariant in its first type parameter could be passed as the argument for a type constructor parameter that assumes its first type parameter to be covariant. This would foil the type system's first-order variance checks: `Seq`'s definition would be invalid if `C` were invariant in its first type parameter.

The remainder of Listing 13 sets up a concrete example that would result in a run-time error if the type application `Seq[A, Cell]` were not ruled out statically.

More generally, a type constructor parameter that does not declare any variance for its parameters does not impose any restrictions on the variance of the parameters of its type argument. However, when either covariance or contravariance is assumed, the corresponding parameters of the type argument must have the same variance.

7. Leveraging Scala's implicits

In this section we discuss how the introduction of type constructor polymorphism has made Scala's support for implicit arguments more powerful. Implicits have been implemented in Scala since version 1.4. They are the minimal extension to an object-oriented language so that Haskell's type classes [56] can be encoded [41].

We first show how to improve the example from Section 3 using implicits, so that clients of `Iterable` no longer need to supply the correct instance of `Buildable[C]`. Since there generally is only one instance of `Buildable[C]` for a particular type constructor `C`, it becomes quite tedious to supply it as an argument whenever calling one of `Iterable`'s methods that requires it.

Fortunately, Scala's implicits can be used to shift this burden to the compiler. It suffices to add the **`implicit`** keyword to the parameter list that contains the `b: Buildable[C]` parameter, and to the `XXXIsBuildable` objects. With this change, which is sketched in Listing 14, callers (such as in

```
trait Seq[+A, C[+X]] { self: C[A] ⇒
  def lift[B >: A]: C[B] = this
}

class Cell[A] extends
    Seq[A, Cell] { // the only (static) error
  private var cell: A = _
  def set(x: A) = cell = x
  def get: A = cell
}

class Top
class Ext extends Top {
  def bar() = println("bar")
}

val exts: Cell[Ext] = new Cell[Ext]
val tops: Cell[Top] = exts.lift[Top]
tops.set(new Top)
exts.get.bar()  // method not found error, if
          // the above static error is ignored
```

Listing 13. Example of unsoundness if higher-order variance annotations are not enforced.

the example of Listing 8) typically do not need to supply this argument.

In the rest of this section we explain this feature in order to illustrate the interaction with type constructor polymorphism. With the introduction of type constructor polymorphism, our encoding of type classes is extended to constructor classes, such as `Monad`, as discussed in Section 7.3. Moreover, our encoding exceeds the original because we integrate type constructor polymorphism with subtyping, so that we can abstract over bounds. This would correspond to abstracting over type class contexts, which is not supported in Haskell [29, 32, 34, 15]. Section 7.3 discusses this in more detail.

7.1 Introduction to implicits

The principal idea behind implicit parameters is that arguments for them can be left out from a method call. If the arguments corresponding to an implicit parameter section are missing, they are inferred by the Scala compiler.

Listing 15 introduces implicits by way of a simple example. It defines an abstract class of monoids and two concrete implementations, `StringMonoid` and `IntMonoid`. The two implementations are marked with an **`implicit`** modifier.

Listing 16 implements a `sum` method, which works over arbitrary monoids. `sum`'s second parameter is marked **`implicit`**. Note that `sum`'s recursive call does not need to pass along the `m` implicit argument.

The actual arguments that are eligible to be passed to an implicit parameter include all identifiers that are marked

```
trait Iterable[T] {
  def map[U](f: T ⇒ U)
            (implicit b: Buildable[Container
    ]): Container[U]
    = mapTo[U, Container](f)
    // no need to pass b explicitly
    // similar for other methods
}

implicit object ListBuildable
                extends Buildable[List]{...}
implicit object OptionBuildable
                extends Buildable[Option]{..}

// client code (previous example, using
    succinct function syntax):
val ages: List[Int]
= bdays.flatMapTo[Int, List]{_.map{toYrs(_)}}
```

Listing 14. Snippet: leveraging implicits in `Iterable`

```
abstract class Monoid[T] {
  def add(x: T, y: T): T
  def unit: T
}

object Monoids {
  implicit object stringMonoid
                  extends Monoid[String] {
    def add(x: String, y: String): String
      = x.concat(y)
    def unit: String = ""
  }
  implicit object intMonoid
                  extends Monoid[Int] {
    def add(x: Int, y: Int): Int
      = x + y
    def unit: Int = 0
  }
}
```

Listing 15. Using implicits to model monoids

```
def sum[T](xs: List[T])(implicit m: Monoid[T
    ]): T
  = if(xs.isEmpty) m.unit else m.add(xs.head,
     sum(xs.tail))
```

Listing 16. Summing lists over arbitrary monoids

```
class  Ord a  where
  (<=) :: a → a → Bool

instance Ord Date where
  (<=)     = ...

max     :: Ord a ⇒ a → a → a
max x y = if x <= y then y else x
```

Listing 17. Using type classes to overload <= in Haskell

```
trait Ord[T] {
  def <= (other: T): Boolean
}

import java.util.Date

implicit def dateAsOrd(self: Date)
  = new Ord[Date] {
    def <= (other: Date) = self.equals(other)
                        || self.before(other)
  }

def max[T <% Ord[T]](x: T, y: T): T
  = if(x <= y) y else x
```

Listing 18. Encoding type classes using Scala's implicits

implicit, and that can be accessed at the point of the method call without a prefix. For instance, the scope of the `Monoids` object can be opened up using an import statement, such as **import** `Monoids._` This makes the two implicit definitions of `stringMonoid` and `intMonoid` eligible to be passed as implicit arguments, so that one can write:

```
sum(List("a", "bc", "def"))
sum(List(1, 2, 3))
```

These applications of `sum` are equivalent to the following two applications, where the formerly implicit argument is now given explicitly.

```
sum(List("a", "bc", "def"))(stringMonoid)
sum(List(1, 2, 3))(intMonoid)
```

If there are several eligible arguments that match an implicit parameter's type, a most specific one will be chosen using the standard rules of Scala's static overloading resolution. If there is no unique most specific eligible implicit definition, the call is ambiguous and will result in a static error.

7.2 Encoding Haskell's type classes with implicits

Haskell's type classes have grown from a simple mechanism that deals with overloading [56], to an important tool in dealing with the challenges of modern software engineering. Its success has prompted others to explore similar features in Java [57].

7.2.1 An example in Haskell

Listing 17 defines a simplified version of the well-known `Ord` type class. This definition says that if a type `a` is in the `Ord` *type class*, the function <= with type `a → a → Bool` is available. The *instance declaration* **instance** `Ord Date` gives a concrete implementation of the <= operation on `Date`'s and thus adds `Date` as an *instance* to the `Ord` type class. To constrain an abstract type to instances of a type class, *contexts* are employed. For example, `max`'s signature constrains `a` to be an instance of `Ord` using the context `Ord a`, which is separated from the function's type by a ⇒.

Conceptually, a context that constrains a type `a`, is translated into an extra parameter that supplies the implementations of the type class's methods, packaged in a so-called "method dictionary". An instance declaration specifies the contents of the method dictionary for this particular type.

7.2.2 Encoding the example in Scala

It is natural to turn a type class into a class, as shown in Listing 18. Thus, an instance of that class corresponds to a method dictionary, as it supplies the actual implementations of the methods declared in the class. The instance declaration **instance** `Ord Date` is translated into an implicit method that converts a `Date` into an `Ord[Date]`. An object of type `Ord[Date]` encodes the method dictionary of the `Ord` type class for the instance `Date`.

Because of Scala's object-oriented nature, the creation of method dictionaries is driven by member selection. Whereas the Haskell compiler selects the right method dictionary fully automatically, this process is triggered by calling missing methods on objects of a type that is an instance (in the Haskell sense) of a type class that does provide this method. When a type class method, such as <=, is selected on a type `T` that does not define that method, the compiler searches an implicit value that converts a value of type `T` into a value that does support this method. In this case, the implicit method `dateAsOrd` is selected when `T` equals `Date`.

Note that Scala's scoping rules for implicits differ from Haskell's. Briefly, the search for an implicit is performed locally in the scope of the method call that triggered it, whereas this is a global process in Haskell.

Contexts are another trigger for selecting method dictionaries. The `Ord a` context of the `max` method is encoded as a view bound `T <% Ord[T]`, which is syntactic sugar for an implicit parameter that converts the bounded type to its view bound. Thus, when the `max` method is called, the compiler must find the appropriate implicit conversion. Listing 19 removes this syntactic sugar, and Listing 20 goes even further and makes the implicits explicit. Clients would then have

```
def max[T](x: T, y: T)
          (implicit conv: T ⇒ Ord[T]): T
  = if(x <= y) y else x
```

Listing 19. Desugaring view bounds

```
def max[T](x: T, y: T)(c: T ⇒ Ord[T]): T
  = if(c(x).<=(y)) y else x
```

Listing 20. Making implicits explicit

to supply the implicit conversion explicitly: `max(dateA, dateB)(dateAsOrd)`.

7.2.3 Conditional implicits

By defining implicit methods that themselves take implicit parameters, Haskell's conditional instance declarations can be encoded:

```
instance Ord a ⇒ Ord (List a) where
  (<=)     = ...
```

This is encoded in Scala as:

```
implicit def listAsOrd[T](self: List[T])(
    implicit v: T ⇒ Ord[T]) =
  new Ord[List[T]] {
    def <= (other: List[T]) = // compare
    elements in self and other
  }
```

Thus, two lists with elements of type `T` can be compared as long as their elements are comparable.

Type classes and implicits both provide ad-hoc polymorphism. Like parametric polymorphism, this allows methods or classes to be applicable to arbitrary types. However, parametric polymorphism implies that a method or a class is truly indifferent to the actual argument of its type parameter, whereas ad-hoc polymorphism maintains this illusion by selecting different methods or classes for different actual type arguments.

This ad-hoc nature of type classes and implicits can be seen as a retroactive extension mechanism. In OOP, virtual classes [48, 20] have been proposed as an alternative that is better suited for retroactive extension. However, ad-hoc polymorphism also allows types to drive the selection of functionality as demonstrated by the selection of (implicit) instances of `Buildable[C]` in our `Iterable` example[6]. `Buildable` clearly could not be truly polymorphic in its parameter, as that would imply that there could be one `Buildable` that knew how to supply a strategy for building any type of container.

```
class Monad m where
  (>>=) :: m a → (a → m b) → m b

data (Ord a) ⇒ Set a = ...

instance Monad Set where
  -- (>>=) :: Set a → (a → Set b) → Set b
```

Listing 21. `Set` cannot be made into a `Monad` in Haskell

```
trait Monad[A, M[X]] {
  def >>= [B](f: A ⇒ M[B]): M[B]
}
```

Listing 22. `Monad` in Scala

```
trait BoundedMonad[A <: Bound[A], M[X <: Bound[
    X]], Bound[X]] {
  def >>= [B <: Bound[B]](f: A ⇒ M[B]): M[B]
}

trait Set[T <: Ord[T]]

implicit def SetIsBoundedMonad[T <: Ord[T]](
  s: Set[T]): BoundedMonad[T, Set, Ord] = ...
```

Listing 23. `Set` as a `BoundedMonad` in Scala

7.3 Exceeding type classes

As shown in Listing 21, Haskell's `Monad` abstraction [55] does not apply to type constructors with a constrained type parameter, such as `Set`, as explained below. Resolving this issue in Haskell is an active research topic [15, 16, 29].

In this example, the `Monad` abstraction[7] does not accommodate constraints on the type parameter of the `m` type constructor that it abstracts over. Since `Set` is a type constructor that constrains its type parameter, it is not a valid argument for `Monad`'s `m` type parameter: `m a` is allowed for any type `a`, whereas `Set a` is only allowed if `a` is an instance of the `Ord` type class. Thus, passing `Set` as `m` could lead to violating this constraint.

For reference, Listing 22 shows a direct encoding of the `Monad` type class. To solve the problem in Scala, we generalise `Monad` to `BoundedMonad` in Listing 23 to deal with bounded type constructors. Finally, the encoding from Section 7.2 is used to turn a `Set` into a `BoundedMonad`.

[6] Java's static overloading mechanism is another example of ad-hoc polymorphism.

[7] In fact, the main difference between our `Iterable` and Haskell's `Monad` is spelling.

8. Related Work

8.1 Roots of our kinds

Since the seminal work of Girard and Reynolds in the early 1970's, fragments of the higher-order polymorphic lambda calculus or System F_ω [23, 50, 7] have served as the basis for many programming languages. The most notable example is Haskell [28], which has supported higher-kinded types for over 15 years [27].

Although Haskell has higher-kinded types, it eschews subtyping. Most of the use-cases for subtyping are subsumed by type classes, which handle overloading systematically [56]. However, it is not (yet) possible to abstract over class contexts [29, 32, 34, 15]. In our setting, this corresponds to abstracting over a type that is used as a bound, as discussed in Section 7.3.

The interaction between higher-kinded types and subtyping is a well-studied subject [13, 12, 10, 49, 17]. As far as we know, none of these approaches combine bounded type constructors, subkinding, subtyping *and* variance, although all of these features are included in at least one of them. A similarity of interest is Cardelli's notion of power types [11], which corresponds to our bounds-tracking kind `*(L, U)`.

In summary, the presented type system can be thought of as the integration of an object-oriented system with Polarized F^{ω}_{sub} [52], Cardelli's power type, and subkinding. Subkinding is based on interval inclusion and the transposition of subtyping of dependent function types [4] to the level of kinds.

8.2 Type constructor polymorphism in OOPL's

Languages with virtual types or virtual classes, such as gbeta [20], can encode type constructor polymorphism through abstract type members. The idea is to model a type constructor such as `List` as a simple abstract type that has a type member describing the element type. Since Scala has virtual types, `List` could also be defined as a class with an abstract type member instead of as a type-parameterised class:

```
abstract class List { type Elem }
```

Then, a concrete instantiation of `List` could be modelled as a type refinement, as in `List{type Elem = String}`. The crucial point is that in this encoding `List` is a type, not a type constructor. So first-order polymorphism suffices to pass the `List` constructor as a type argument or an abstract type member refinement.

Compared to type constructor polymorphism, this encoding has a serious disadvantage, as it permits the definition of certain accidentally empty type abstractions that cannot be instantiated to concrete values later on. By contrast, type constructor polymorphism has a *kind soundness* property that guarantees that well-kinded type applications never result in nonsensical types.

Type constructor polymorphism has recently started to trickle down to object-oriented languages. Cremet and Altherr's work on extending Featherweight Generic Java with higher-kinded types [18] partly inspired the design of our syntax. However, since they extend Java, they do not model type members and path-dependent types, definition-site variance, or intersection types. They do provide direct support for anonymous type constructors. Furthermore, although their work demonstrates that type constructor polymorphism can be integrated into Java, they only provide a prototype of a compiler and an interpreter. However, they have developed a mechanised soundness proof and a pencil-and-paper proof of decidability.

Finally, we briefly mention OCaml and C++. C++'s template mechanism is related, but, while templates are very flexible, this comes at a steep price: they can only be type-checked after they have been expanded. Recent work on "concepts" alleviates this [25].

In OCaml (as in ML), type constructors are first-order. Thus, although a type of, e.g., kind $\star \to \star \to \star$ is supported, types of kind $(\star \to \star) \to \star \to \star$ cannot be expressed directly. However, ML dialects that support applicative functors, such as OCaml and Moscow ML, can encode type constructor polymorphism in much the same way as languages with virtual types.

9. Conclusion

Genericity is a proven technique to reduce code duplication in object-oriented libraries, as well as making them easier to use by clients. The prime example is a collections library, where clients no longer need to cast the elements they retrieve from a generic collection.

Unfortunately, though genericity is extremely useful, the first-order variant is self-defeating in the sense that abstracting over proper types gives rise to type constructors, which cannot be abstracted over. Thus, by using genericity to reduce code duplication, other kinds of boilerplate arise. Type constructor polymorphism allows to further eliminate these redundancies, as it generalises genericity to type constructors.

As with genericity, most use cases for type constructor polymorphism arise in library design and implementation, where it provides more control over the interfaces that are exposed to clients, while reducing code duplication. Moreover, clients are not exposed to the complexity that is inherent to these advanced abstraction mechanisms. In fact, clients *benefit* from the more precise interfaces that can be expressed with type constructor polymorphism, just like genericity reduced the number of casts that clients of a collections library had to write.

We implemented type constructor polymorphism in Scala 2.5. The essence of our solution carries over easily to Java, see Altherr et al. for a proposal [3].

Finally, we have only reported on one of several applications that we have experimented with. Embedded domain specific languages (DSL's) [14] are another promising appli-

cation area of type constructor polymorphism. We are currently applying these ideas to our parser combinator library, a DSL for writing EBNF grammars in Scala [39]. Hofer, Ostermann et al. are investigating similar applications [26], which critically rely on type constructor polymorphism.

Acknowledgments

The authors would like to thank Dave Clarke, Marko van Dooren, Burak Emir, Erik Ernst, Bart Jacobs, Andreas Rossberg, Jan Smans, and Lex Spoon for their insightful comments and interesting discussions. We also gratefully acknowledge the Scala community for providing a fertile testbed for this research. Finally, we thank the reviewers for their detailed comments that helped us improve the paper. An older version of this paper was presented at the MPOOL workshop [38].

The first author is supported by a grant from the Flemish IWT. Part of the reported work was performed during a 3-month stay at EPFL.

References

[1] M. Abadi and L. Cardelli. A theory of primitive objects: Second-order systems. *Sci. Comput. Program.*, 25(2-3):81–116, 1995.

[2] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. *Inf. Comput.*, 125(2):78–102, 1996.

[3] P. Altherr and V. Cremet. Adding type constructor parameterization to Java. Accepted to the workshop on Formal Techniques for Java-like Programs (FTfJP'07) at the European Conference on Object-Oriented Programming (ECOOP), 2007.

[4] D. Aspinall and A. B. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1-2):273–309, 2001.

[5] G. M. Bierman, E. Meijer, and W. Schulte. The essence of data access in Comega. In A. P. Black, editor, *ECOOP*, volume 3586 of *Lecture Notes in Computer Science*, pages 287–311. Springer, 2005.

[6] G. Bracha. Executable grammars in Newspeak. *Electron. Notes Theor. Comput. Sci.*, 193:3–18, 2007.

[7] K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Inf. Comput.*, 85(1):76–134, 1990.

[8] K. B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In E. Jul, editor, *ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 523–549. Springer, 1998.

[9] K. B. Bruce, A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In W. G. Olthoff, editor, *ECOOP*, volume 952 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 1995.

[10] P. S. Canning, W. R. Cook, W. L. Hill, W. G. Olthoff, and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *FPCA*, pages 273–280, 1989.

[11] L. Cardelli. Structural subtyping and the notion of power type. In *POPL*, pages 70–79, 1988.

[12] L. Cardelli. Types for data-oriented languages. In J. W. Schmidt, S. Ceri, and M. Missikoff, editors, *EDBT*, volume 303 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1988.

[13] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, 1985.

[14] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated. In Z. Shao, editor, *APLAS*, volume 4807 of *Lecture Notes in Computer Science*, pages 222–238. Springer, 2007.

[15] M. Chakravarty, S. L. P. Jones, M. Sulzmann, and T. Schrijvers. Class families, 2007. On the GHC Developer wiki, http://hackage.haskell.org/trac/ghc/wiki/TypeFunctions/ClassFamilies.

[16] M. M. T. Chakravarty, G. Keller, S. L. P. Jones, and S. Marlow. Associated types with class. In J. Palsberg and M. Abadi, editors, *POPL*, pages 1–13. ACM, 2005.

[17] A. B. Compagnoni and H. Goguen. Typed operational semantics for higher-order subtyping. *Inf. Comput.*, 184(2):242–297, 2003.

[18] V. Cremet and P. Altherr. Adding type constructor parameterization to Java. *Journal of Object Technology*, 7(5):25–65, June 2008. Special Issue: Workshop on FTfJP, ECOOP 07. http://www.jot.fm/issues/issue_2008_06/article2/.

[19] B. Emir, A. Kennedy, C. V. Russo, and D. Yu. Variance and generalized constraints for C# generics. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2006.

[20] E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

[21] E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001.

[22] J. Gibbons and G. Jones. The under-appreciated unfold. In *ICFP*, pages 273–279, 1998.

[23] J. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. Thèse d'État, Paris VII, 1972.

[24] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[25] D. Gregor, J. Järvi, J. G. Siek, B. Stroustrup, G. D. Reis, and A. Lumsdaine. Concepts: linguistic support for generic programming in C++. In P. L. Tarr and W. R. Cook, editors, *OOPSLA*, pages 291–310. ACM, 2006.

[26] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In Y. Smaragdakis and J. Siek, editors, *GPCE*. ACM, 2008. To appear.

[27] P. Hudak, J. Hughes, S. L. P. Jones, and P. Wadler. A history of Haskell: being lazy with class. In B. G. Ryder and B. Hailpern, editors, *HOPL*, pages 1–55. ACM, 2007.

[28] P. Hudak, S. L. P. Jones, P. Wadler, B. Boutel, J. Fairbairn, J. H. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, R. B. Kieburtz, R. S. Nikhil, W. Partain, and J. Peterson. Report on the programming language Haskell, a non-strict, purely functional language. *SIGPLAN Notices*, 27(5):R1–R164, 1992.

[29] J. Hughes. Restricted datatypes in Haskell. Technical Report UU-CS-1999-28, Department of Information and Computing Sciences, Utrecht University, 1999.

[30] G. Hutton and E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[31] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[32] M. P. Jones. constructor classes & "set" monad?, 1994. http://groups.google.com/group/comp.lang.functional/msg/e10290b2511c65f0.

[33] M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Funct. Program.*, 5(1):1–35, 1995.

[34] E. Kidd. How to make data.set a monad, 2007. http://www.randomhacks.net/articles/2007/03/15/data-set-monad-haskell-macros.

[35] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001.

[36] E. Meijer. There is no impedance mismatch: (language integrated query in Visual Basic 9). In P. L. Tarr and W. R. Cook, editors, *OOPSLA Companion*, pages 710–711. ACM, 2006.

[37] E. Meijer. Confessions of a used programming language salesman. In R. P. Gabriel, D. F. Bacon, C. V. Lopes, and G. L. S. Jr., editors, *OOPSLA*, pages 677–694. ACM, 2007.

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

[39] A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2008. http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html.

[40] A. Moors, F. Piessens, and M. Odersky. Safe type-level abstraction in Scala. In *Proc. FOOL '08*, Jan. 2008. http://fool08.kuis.kyoto-u.ac.jp/.

[41] M. Odersky. Poor man's type classes, July 2006. Talk at IFIP WG 2.8, Boston.

[42] M. Odersky. *The Scala Language Specification, Version 2.6*. EPFL, Nov. 2007. http://www.scala-lang.org/docu/files/ScalaReference.pdf.

[43] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, L. Spoon, E. Stenman, and M. Zenger. An Overview of the Scala Programming Language (2. edition). Technical report, 2006.

[44] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In L. Cardelli, editor, *ECOOP*, volume 2743 of *Lecture Notes in Computer Science*, pages 201–224. Springer, 2003.

[45] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.

[46] M. Odersky, C. Zenger, and M. Zenger. Colored local type inference. In *POPL*, pages 41–53, 2001.

[47] M. Odersky and M. Zenger. Scalable component abstractions. In R. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 41–57. ACM, 2005.

[48] H. Ossher and W. H. Harrison. Combination of inheritance hierarchies. In *OOPSLA*, pages 25–40, 1992.

[49] B. C. Pierce and M. Steffen. Higher-order subtyping. *Theor. Comput. Sci.*, 176(1-2):235–282, 1997.

[50] J. C. Reynolds. Towards a theory of type structure. In B. Robinet, editor, *Symposium on Programming*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[51] T. Sheard. Type-level computation using narrowing in Ωmega. *Electr. Notes Theor. Comput. Sci.*, 174(7):105–128, 2007.

[52] M. Steffen. *Polarized Higher-Order Subtyping*. PhD thesis, Universität Erlangen-Nürnberg, 1998.

[53] K. K. Thorup and M. Torgersen. Unifying genericity - combining the benefits of virtual types and parameterized classes. In R. Guerraoui, editor, *ECOOP*, volume 1628 of *Lecture Notes in Computer Science*, pages 186–204. Springer, 1999.

[54] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[55] P. Wadler. Monads for functional programming. In J. Jeuring and E. Meijer, editors, *Advanced Functional Programming*, volume 925 of *Lecture Notes in Computer Science*, pages 24–52. Springer, 1995.

[56] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[57] S. Wehr, R. Lämmel, and P. Thiemann. JavaGI : Generalized interfaces for Java. In E. Ernst, editor, *ECOOP*, volume 4609 of *Lecture Notes in Computer Science*, pages 347–372. Springer, 2007.