Generics of a Higher Kind

Adriaan Moors[1], Frank Piessens[1], and Martin Odersky[2]

[1] K.U. Leuven
{adriaan, frank}@cs.kuleuven.be
[2] 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 types. This paper presents the design and implementation of the resulting type constructor polymorphism. It combines type constructor polymorphism with implicit parameters to yield constructs similar to, and at times more expressive than, Haskell's constructor type classes. The paper also studies interactions with other object-oriented language constructs, and discusses the gains in expressiveness.

1 Introduction

First-order parametric polymorphism is now a standard element of statically typed programming languages. Starting with System F [20,42] and functional programming languages, the constructs have found their way into mainstream languages such as Java and C#. 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. [26], 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. It makes it possible to abstract over types, yielding *type constructors* such as `List`. However, the resulting type constructors cannot themselves be abstracted over. For instance, one cannot usually pass a type constructor such as `List` as a type argument to another type constructor. It turns out that this restriction prevents the formulation of some quite natural abstractions and thus leads to unnecessary duplication of code. We provide several examples of such abstractions in this paper.

More generality can be achieved by passing to higher-order *type constructor polymorphism*. The generalisation to higher-order polymorphism has been a natural step in lambda calculus [20,42,7] and it has also influenced the design of functional programming languages. For instance, the Haskell programming language [23] supports type

constructor polymorphism, which is also integrated with its type class concept [28]. This generalisation to types that abstract over types that abstract over types ("higher-kinded types") has many practical applications. For example, comprehensions [45], parser combinators [25,30], and recent work on embedded Domain Specific Languages (DSL's) [13] critically rely on higher-kinded types.

The same needs – as well as more specific ones – arise in object-oriented programming. LINQ introduced direct support for comprehensions on .NET[5,33], Scala has had a similar feature from the start, and Java 5 introduced a lightweight variation. 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 [34] that allows users to express parsers directly in Scala, in a notation that closely resembles EBNF.

In this paper, we study the design and implementation of type constructor polymorphism in the Scala programming language. These higher-order generics have been available in Scala from version 2.5, which was made available as a public distribution in May 2007. We motivate why abstracting over type constructors is useful and how it can avoid code redundancies. We develop a system of kinds for characterising types and type constructors. Kinds express which types or type constructors are admissible instances at an abstraction point; they play the same role for types that types play for values. Our kinds capture three different aspects of a type or type constructor: its shape, its lower and/or upper bounds, and its variance.

We then show how type constructor polymorphism can be combined with Scala's implicit parameters to provide expressiveness analogous to type constructor classes in Haskell. In fact, the combination of higher-kinded types, implicits, and subtyping lets us express concepts such as bounded monads that are beyond the reach of standard type constructor classes.

Languages with virtual types or virtual classes 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. Scala belongs to this category of languages. For instance, in Scala you could also define `List` 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 an type argument or an abstract type instance.

Compared to type constructor polymorphism, this encoding has three disadvantages, however. First, it is considerably more verbose. Second, it requires the definition of named members representing the element types, which induces the risk of accidental name clashes in class inheritance hierarchies. Third, the encoding permits the definition of certain nonsensical type abstractions that cannot be instantiated to concrete values later on. By contrast, type constructor polymorphism has a *kind soundness* property which guarantees that well-kinded type applications never result in nonsensical types. These are three reasons that argue in favour of including type constructor polymorphism in object-oriented programming.

The main contributions of this paper are as follows:

- We describe an implementation of type constructor polymorphism in a widely used object-oriented language.
- We develop a kind-system that captures both lower and upper bounds and variances of types.
- We combine higher-kinded types with Scala's implicit parameters to provide expressiveness analogous Haskell's type constructor classes.
- We show that the combination of higher-kinded types, implicit parameters, and subtyping can express concepts such as a bounded monad, which cannot be expressed by classical type constructor classes.
- We discuss an encoding of type constructor polymorphism using Scala's abstract type members.
- We formulate the kind soundness property of type constructor polymorphism, and explain why it gets lost in the encoding to abstract type members.

The rest of this paper is structured as follows. Section 2 demonstrates that type constructor polymorphism reduces boilerplate that arises from the use of genericity. We start out with a simple example, and extend it to a full implementation of the comprehensions fragment of `Iterable`. Section 3 shows the utility of implicits in OOP, as well as how they can be used to encode Haskell's constructor type classes. Section 4 further extends our kind system so that we can safely abstract over type constructors with bounded type parameters. We then apply this generalisation to `Iterable` and our encoding of type classes. Section 5 relates the functional and object-oriented styles of building abstractions, and introduces the notion of *kind soundness*. Section 6 illustrates the need for higher-order variance annotations, which are required for type soundness. Finally, we summarise related work in Section 7 and conclude in Section 8.

2 Reducing Code Duplication with Type Constructor Polymorphism

In this section, we illustrate the benefits of generalising genericity to type constructor polymorphism using the well-known `Iterable` abstraction. We begin with a simple example, which is due to Alexander Spoon, but we will extend it to more realistic proportions in section 2.2.

Figure 1 shows a Scala [38] implementation of the trait `Iterable[T]`, which is an abstract class that supports mixin composition. 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`. `remove` is implemented in terms of `filter`, as it simply inverts the meaning of the predicate.

Naturally, when filtering a list, we expect to again receive a list. Thus, `List` overrides `filter` to refine its result type covariantly. For brevity, we omitted `List`'s subclasses, which implement this method. For consistency, `remove` should have the same result type, but the only way to achieve this is by overriding it as well. The resulting

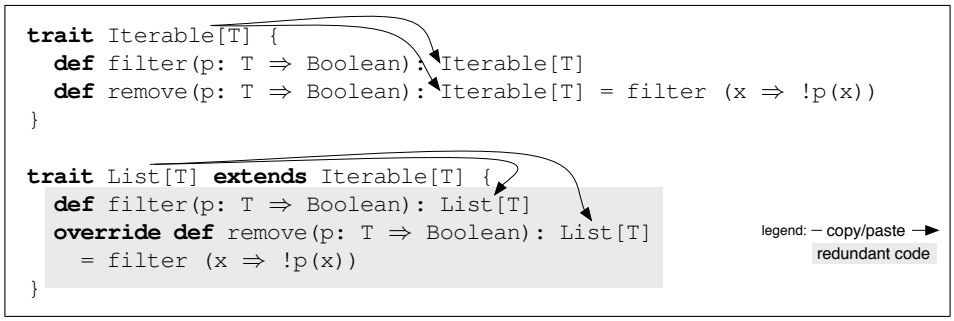

Fig. 1. Limitations of Genericity

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`.

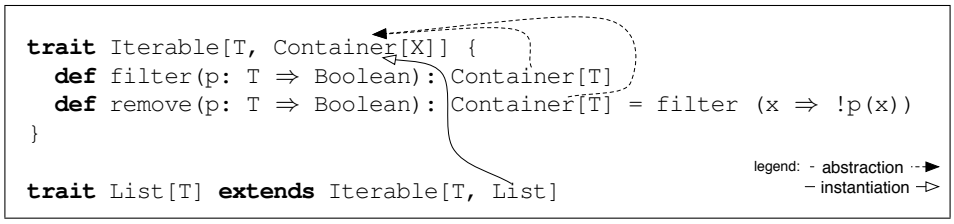

Fig. 2. Removing Code Duplication

Our solution, depicted in Fig. 2, is to abstract over the type constructor that represents the container of the result of `filter` and `remove`. Our 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. Note that the name of this higher-order type parameter (`X`) is not relevant here.

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, we could also have used a construct like Bruce's `MyType` [9]. However, this scheme breaks down in more complex cases, as we will demonstrate in Section 2.2. First, we introduce type constructor polymorphism in more detail.

2.1 Type constructors and kinds

Type constructor polymorphism generalises genericity so that we can abstract over type constructors (such as `List`) as well as proper types (such as `Int` or `List[Int]`). To

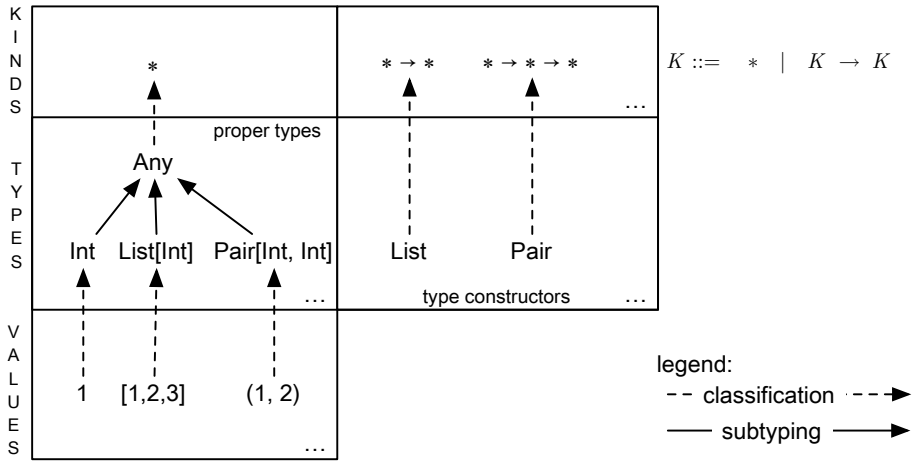

Fig. 3. Diagram of levels

distinguish proper types from type constructors, we use "kinds" (a term borrowed from functional programming). Kinds are to types as types are to values. This divides our language into three levels: at the bottom, we have objects (our values), as depicted in Fig. 3. Objects are classified by types, which reside in the next level. Finally, types are classified by kinds.

Unlike types, kinds are purely structural: they simply reflect the kinds of the type parameters that a type expects. Since proper types all take the same number of type parameters (i.e., none), they are classified by the same kind, $\star$. To classify type constructors, we use a kind *constructor* `From` $\rightarrow$ `To`, which abstracts over the kinds `From` and `To`. `From` is the kind of the expected type argument and `To` is the kind of the type that results from applying the type constructor to an argument.

For example, **`class`** `List[T]` gives rise to a type constructor `List` that is classified by the kind $\star \rightarrow \star$, as applying `List` to a proper type yields a proper type. Note that, since kinds are structural, given e.g., **`class`** `Animal[FoodType]`, `Animal` has the exact same kind as `List`.

Our initial[3] model of the level of kinds can be described using the following grammar:

$$K ::= \quad * \quad | \quad K \rightarrow K$$

The rules that define the well-formedness of types in a language without type constructor polymorphism, correspond to the rules that assign a kind $\star$ to a type. Our extensions generalise this to the notion of kind checking, which is to types as type checking is to values and expressions.

[3] In Section 4, we will extend this model with support for bounds, and Section 6 describes the impact of variance on the level of kinds.

```
trait Iterable[T] {
  type Container[X]

  def filter(p: T ⇒ Boolean): Container[T]
}
```

Listing 1. `Iterable` with an abstract type constructor member

A class, or an unbounded type parameter or abstract type member receives the kind $K' \rightarrow *$ if it has one type parameter with kind K'. For bounded type parameters or abstract members, the kind $K' \rightarrow K$ is assigned, where K corresponds to the bound. We use currying to generalise this scheme to deal with multiple type parameters. The type application `T[T']` has the kind K if T has kind $K' \rightarrow K$, and T' is classified by the kind K'.

Finally, the syntactical impact of extending Scala with type constructor polymorphism is minor. Before, only classes and type aliases could declare formal type parameters, whereas this has now been extended to include type parameters and abstract type members. Figure 2 already introduced the notation for type constructor parameters, and Listing 1 completes the picture with an alternative formulation of our running example using an abstract type constructor member.

2.2 Improving Iterable

In this section we design and implement the abstraction that underlies comprehensions. 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 introduced these on the .NET platform as `Select`, `Where`, and `SelectMany` [32].

These methods interpret a user-supplied function in different ways 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 that 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.

Thus, if we can factor out iterating over a collection as well as producing a new one, these methods can be implemented in `Iterable` once and for all. Listing 2 shows the well-known `Iterator` abstraction that encapsulates iterating over a collection, as well as our `Builder` abstraction, which may be thought of as its dual.

`Builder` abstracts over the type constructor that represents the collection that it builds, as well as over the type of the elements. The `+=` method is used to supply these elements in the order in which they should appear in the collection. The collection itself is returned by `finalise`. For example, a `Builder[List, Int]` can be thought of as a `ListBuffer[Int]`, as both can be used to create a `List[Int]` by supplying its elements in turn.

```
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 2. `Builder` and `Iterator`

With these abstractions in place, we turn to Listing 3, and show how an even more flexible trio `mapTo`/`filterTo`/`flatMapTo` is implemented. 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.

As iterating over a collection is orthogonal to building a collection, the `build` method from `Buildable` does not belong in `Iterable`. In other words, it is not necessary to be able to build a collection in order to simply iterate over its elements. However, more complex operations, such as `mapTo`, do require an instance of `Buildable[C]`. Thus, `Iterable`'s methods that build a collection `C`, take an extra parameter of type `Buildable[C]`. Section 3 will show how an orthogonal feature of Scala can be used to relieve callers from supplying an actual argument for this parameter.

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's the the same type constructor, but it is applied to different type arguments!

Listings 4 and 5 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.

Finally, a brief note on methodology: `Container` is a parameter in `Buildable` because its main characteristic is which container it builds, whereas we use a type member in `Iterable`, as its external clients are generally only interested in the type of its elements. Thus, the `Container` member is more of an internal matter in `Iterable`'s subclassing hierarchy.

Example Suppose we're developing a social networking site, and we want to know the average age of our users. Since users do not have to enter their birthday, we set out with a `List[Option[Date]]`. An `Option[Date]` either holds a date or nothing. Listing 6 shows how to proceed.

First, we introduce a small helper that computes the current age in years from a date of birth. To collect the known ages, we transform an optional date to an optional age using `map` and then collect the results into a list using `flatMapTo`. Note that we use

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

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 buff = b.build[T]
    val elems = elements

    while(elems.hasNext){
      val el = elems.next
      if(p(el)) buff += el
    }
    buff.finalise()
  }
  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 3. `Buildable` and `Iterable`

the more general `flatMapTo`. If we had used `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, we aggregate the results using `reduceLeft`. The full code of the example is available on the paper's homepage[4].

Note that the Scala compiler infers most proper types (we added some annotations to aid understanding), but it does not 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 it cannot be inferred. 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. Furthermore, the next section discusses how the arguments `OptionIsBuildable` and `ListIsBuildable` can be omitted.

3 Leveraging Scala's implicits

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 [36,37] 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 7, callers (such as in the example of Listing 6) typically do not need to supply this argument.

Implicits are one of the major innovations in Scala. They have been implemented since version 1.4 of the language, but so far have not yet been described in a conference or journal publication. One aspect of implicits is that they can encode type classes, as they are found in Haskell. The "overhead" in language concepts to do this encoding is very small, because implicits make use of the standard object-oriented machinery whereas Haskell introduces more specialised concepts. In a C++ context, implicits resemble Siek and Lumsdaine's "concepts" [21].

With the introduction of type constructor polymorphism, our encoding of type classes is extended to constructor classes, such as `Monad`, as discussed in Section 4.4. 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 [24,27,29,14]. Section 4.4 discusses this in more detail.

Listing 8 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.

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

```
object ListIsBuildable 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 4. Building a `List`

```
object OptionIsBuildable 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 UnsupportedOperationException(">1 elements")

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

Listing 5. Building an `Option`

```
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)}(OptionIsBuildable)
                      }(ListIsBuildable)

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

Listing 6. Example: using `Iterable`

```
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 ListIsBuildable extends Buildable[List] { ... }
implicit object OptionIsBuildable extends Buildable[Option] { ... }

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

Listing 7. 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 8. 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 9. Summing lists over arbitrary monoids

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 9 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 **`implicit`**, and that can be accessed at the point of the method call without a prefix. For instance, we can open up the scope of the `Monoids` object 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 we 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 which 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.

3.1 Implicit parameters generalise subtype bounds

Adding a parameter list to a method clearly constrains how that method may be called. For example, since `mapTo` declares a parameter of type `Buildable[C]`, it can only be called when an argument of type `Buildable[C]` can be supplied. Making this parameter list implicit shifts this burden to the compiler, but the constraint remains.

Similarly, subtype bounds on a method's type parameters constrain how this method may be called, as the actual type arguments supplied by the client must meet the declared bounds. Conceptually, this may be thought of as an implicit parameter that represents the coercion that corresponds to the subtype bound.

Thus, both mechanisms can be used to restrict method calls. In both cases, the compiler carries the burden of providing the witness to that constraint. However, implicit parameters are more general than subtype bounds in two ways.

First, the programmer has access to the value that witnesses the constraint. With subtyping, a bound on a type parameter `T` simply results in more information being available on values of type `T`. To use that information, a value of type `T` is needed.

In the case of type constructor parameters, such as `C` in `mapTo`, it is less useful to gain more information about values of type `C[T]` for some type `T`. More concretely, suppose we would use a subtype constraint `C[X] <: Buildable[C]` instead of an implicit parameter. Thus, for any `T`, a `C[T]` is also a `Buildable[C]`. In this case, this results in a Catch-22: we need an instance of `Buildable[C]` to create an object of type `C[T]`, but the only way to acquire a `Buildable[C]` is to coerce a value of type `C[T]`. Moreover, the type `T` needs to be specified even though it is irrelevant for the `Buildable` abstraction.

Using an implicit parameter instead of a bound solves both problems, without increasing the burden on clients of the abstraction. The compiler automatically supplies the right instance of `Buildable[C]`, without the need to specify a type `T` or acquire a value of type `C[T]`.

Second, implicit parameters can constrain other abstract types besides the method's own type parameters. As an example of such a generalised constraint [17], the `map`/`filter`/`flatMap` methods all require an implicit parameter of type `Buildable[Container]`. This effectively constrains the `Container` type constructor member, which is not possible using ordinary subtype bounds.

3.2 Encoding Haskell's type classes with implicits

Haskell's type classes have grown from a simple mechanism that deals with overloading [47], to an important tool in dealing with the challenges of modern software engineering. Its success has prompted others to explore similar features in Java [48].

An example in Haskell Listing 10 defines a simplified version of the well-known `Ord` type class. Essentially, 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

```
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 10. 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 11. Encoding type classes using Scala's implicits

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.

Encoding the example in Scala It is natural to turn a type class into a class, as shown in Listing 11. 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 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

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

Listing 12. Desugaring view bounds

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

Listing 13. Making implicits explicit

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 12 removes this syntactic sugar, and Listing 13 goes even further and makes the implicits explicit. Clients would then have to supply the implicit conversion explicitly: `max(dateA, dateB)(dateAsOrd)`.

Conditional implicits By defining implicit methods that themselves take implicit parameters, we can encode Haskell's conditional instance declarations. For example:

```
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. To ensure that the compiler's search for implicit arguments terminates, the Scala Language Specification defines a contractiveness check for implicit methods [37].

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 retro-active extension mechanism. In OOP, virtual classes [18] have been proposed as an alternative that's better suited retro-active 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[5]. `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.

4 Bounds for Type Constructors

Subtyping and parameterisation are both important sources of polymorphism. We discuss how they interact and how this impacts our model of kinds. To illustrate the advantages of integrating subtyping and type constructors, we extend `Iterable` to deal with bounded collections. In this regard, Scala's type system is more expressive than Haskell's: it is not possible to abstract over type class contexts in Haskell, so that `Set` (a bounded collection) cannot be made into a `Monad` (an `Iterable`) [24,14].

4.1 Tracking bounds in kinds

Given the definition **`class`** `NumericList[T <: Number]`, `NumericList` must not simply be classified as a $\star \rightarrow \star$. Otherwise, the kinding rule for type applications would consider `NumericList[String]` well-kinded, which clearly is not the case.

Thus, we must make information about the bounds on type parameters available to the kinding system. We enrich the definition of $\star$ so that it includes the lower and upper bounds that must be satisfied by the types that it classifies:

$$K ::= \quad *(T , T) \quad | \quad K \rightarrow K$$

This improvement neatly incorporates bounded types into our three-level model of classification. As `Any` represents the top of Scala's subtype lattice, and `Nothing` is the bottom type, we recover $\star$ as `*(Nothing, Any)`, but we will keep using it as a shorthand. Since we predominantly use upper bounds, we abbreviate `*(Nothing, T)` to `*(T)`. Thus, `NumericList` has kind `*(Number)` $\rightarrow \star$.

We further adapt our kinding system to assign the kind `*(T, T)` to a type `T` that previously simply received kind $\star$, so that `String` has kind `*(String, String)`. With these improvements, the illegal application `NumericList[String]` is ruled out.

Unfortunately, even though `Int <: Number`, `NumericList[Int]` is not well-kinded in this system, as `*(Int, Int)` cannot be related to `*(Number)`. To achieve this, we introduce subkinding and kind subsumption.

4.2 Subkinding

A kind `*(S, T)` is a subkind of `*(S', T')` if `S' <: S` and `T <: T'`, which corresponds to the usual notion of interval inclusion. The $\rightarrow$ kind constructor behaves like the type constructor for function types: it is contravariant in the kind of the parameter, and covariant in the kind of the result.

We also allow kind subsumption. That is, if `T: K` and `K` is a subkind of `K'`, then `T : K'`. For instance, `Int` : `*(Int, Int)` implies `Int` : `*(Nothing, Number)`. Thus, `NumericList[Int]` is well-kinded.

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

```
trait NumericList[T <: Number] extends Iterable[T] {
  type Container[X <: Number] = NumericList[X]
}
```

Listing 14. `NumericList`: an illegal subclass of `Iterable`

```
trait Iterable[T, Container[X]]

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

Listing 15. Rephrasing `NumericList` with type parameters

4.3 Bounded Iterable

Given `Iterable`'s definition in Listing 3, it is illegal to subclass it as in Listing 14. If this were allowed, `Iterable`'s abstract type member `Container` would forget the bound on `NumericList`, so that a client could use the method `map[String]` on a `NumericList` to coax it into producing a `NumericList[String]`.

The kinding rules that we introduced earlier, suffice to detect this problem. To make them easier to apply, Listing 15 rephrases the problem using only type parameters (this equivalence is discussed in Section 5).

The type application `Iterable[T, NumericList]` is not well-kinded because `Iterable` declares the kind $\star \rightarrow \star$ for the `Container` parameter, whereas `NumericList` has kind $\star$`(Number)` $\rightarrow \star$, which does not conform to $\star \rightarrow \star$.

To allow subclasses of `Iterable` to declare a bound on the type of elements they accept, `Iterable` must abstract over this bound. Listing 16 generalises the interface of the original `Iterable` from Listing 3. The implementation is not affected by this change. `NumericList` can now be defined as shown in Listing 17.

Again, the client of the collections API is not exposed to the relative complexity of Listing 16. 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 means that they need to learn fewer concepts.

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

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

Listing 16. Essential changes to extend Iterable with support for bounds

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

Listing 17. Safely subclassing `Iterable`

```
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 18. `Set` cannot be made into a `Monad` in Haskell

4.4 Exceeding type classes

As shown fragmentarily in Listing 18, Haskell's `Monad` abstraction [46] does not apply to type constructors with a constrained type parameter, such as `Set`. The reason is similar to why the `NumericList` from the previous section could not be made into a subclass of `Iterable`, unless the latter accommodates bounds from the start. Resolving this issue in Haskell is an active research topic [14,15,24].

In this example, the `Monad` abstraction[6] 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 19 shows a direct encoding of the `Monad` type class. To solve the problem in Scala, we then generalise `Monad` to `BoundedMonad` in Listing 20 to deal with bounded type constructors. Finally, implicits are used to retro-actively turn `Set` into a `BoundedMonad`, as explained in Section 3.2.

5 Embedding in an Object-Oriented Language

Scala supports two styles of abstraction: the functional style uses parameterisation, whereas abstract members represent the object-oriented way. Both styles have differ-

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

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

Listing 19. `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 20. `Set` as a `BoundedMonad` in Scala

ent strengths [8,44,19], and the examples in the previous sections have employed them in a way that is idiomatic to Scala.

It is natural to ask how these styles relate, and whether one style could be used exclusively. To answer this, we show how type constructor polymorphism can be encoded using abstract type members and abstract type member refinement, which are purely object-oriented mechanisms.

Although the encoding works for correct programs, it does not preserve the safety properties of type constructor polymorphism. Certain illegal type applications are encoded as valid Scala programs, which delays the detection of these errors.

By analogy to ordinary soundness, which ensures that a well-typed program does not contain function applications that pass the wrong type of arguments at run time, we use the term "kind soundness" for the property that well-kinded type applications never result in vacuous types. Recently, we developed a purely object-oriented formalism that possesses this property [35].

5.1 Encoding type constructors using abstract type members

At first sight, Scala's abstract type members closely correspond to type parameters, and abstract type member refinement (a restricted form of mixin composition) is the object-oriented counterpart of type application. Abstract type member refinement allows to override abstract type members with concrete ones. The rules for application and abstraction carry over straightforwardly to this approach. In fact, via this correspondence, Scala already provided preliminary support for type constructor polymorphism before our extension. Listing 21 illustrates this encoding using our running example.

5.2 Kind soundness

Listing 22 illustrates the limitations of the encoding. As discussed in Section 4.3, `NumericList` is not a valid subclass of an `Iterable` that does not accommodate bounds. Thus, it must be rejected by the compiler, as the inherited `map` method's result type may apply `NumericList` to any type, but `NumericList` only accepts subtypes of `Number` as its type argument.

The Scala compiler silently accepts this program, even though we could never complete its implementation (at some point we will have to instantiate a `NumericList` for an arbitrary type of elements, and the compiler will catch our mistake).

```
trait TypeFunction1 { type A }

trait Iterable extends TypeFunction1 {
  type Container <: TypeFunction1

  def map[B](f: A ⇒ B): Container{type A = B}
}

trait List extends Iterable { type Container = List }
```

Listing 21. Encoding the `Container` type parameter as an abstract member

```
trait NumericList extends Iterable {
  type A <: Number
  type Container = NumericList // Incorrect, but no error reported!
}
```

Listing 22. `NumericList` eludes the type checker

Note that this indulgence does not imply *type* unsoundness, as these erroneous types cannot be instantiated. Nonetheless, we regard it as a shortcoming of the compiler that these degenerate types are allowed to slip by unnoticed. Even though they are prevented from being instantiated, they could be unmasked earlier.

To motivate this desire for early detection of these inconsistencies, consider the analogy with abstract classes. Suppose classes would be allowed to be abstract implicitly, so that accidental abstract classes would not be discovered until a client attempts to instantiate them. However, this situation is considered undesirable by most languages, so that an abstract class must be marked as such explicitly. This eliminates the possibility that the programmer simply forgot to implement a method.

Not detecting erroneous type applications, which manifest themselves as intersection types that unexpectedly do not have any instances, has the same effect as allowing any class to be abstract implicitly: eventually, the error is detected, but it could have been signalled earlier. Even though other uses of intersection types might sensibly result in empty types, we do not consider this to be one of them.

This kind unsoundness has its roots in the νObj calculus [39], which allows abstract type members to be refined *covariantly*, thus `NumericList <: TypeFunction1`. In this case, this means that `NumericList <: TypeFunction1` holds, whereas the corresponding subkinding judgement $\star$`(Number)` $\rightarrow \star <: \star \rightarrow \star$ does not hold.

Related work seems to deviate from νObj's design, although making a precise comparison is complicated by the differences in features supported by the various approaches. In the notation of Cardelli [11], the main two types are classified as follows:

```
NumericList : ALL[A::POWER[Number]] TYPE
TypeFunction1 : ALL[X::TYPE] TYPE
```

Cardelli does not define subkinding for these kinds, but does define subtyping for polymorphic functions ("All[X::K]B <: All[X::K']B' if K'<::K (where <:: denotes a subkind relation[, ...]), and B<:B' under the assumption that X::K"'). It seems reasonable to lift this rule (which deals with functions that take a type to yield a *value*) to the level of kinds, which results in our rule that deals with functions that take a type to yield a *type*.

Similarly, in the notation of Compagnoni and Goguen [16]:

```
NumericList : Pi A <: Number : *. *
TypeFunction1 : Pi X <: T* : *. *
```

Although the authors require these bounds to be equal for the kinds to be comparable (their treatment does not include subkinding), we generalise based on the same observation as the previous paragraph, but using a slight different source of inspiration. Namely, Full System $F_{<:}$'s [12] rule that deals with bounded quantification at the value level (Sub Forall) also requires *contravariance* for the bounds of the quantifier.

We recover early error detection in Scalina [35], a purely object-oriented calculus, by differentiating covariant and contravariant members, instead of assuming they all behave covariantly. This distinction corresponds to the fact that some members abstract over input, whereas others represent the output of the abstraction. Input members should behave contravariantly, like the types of a method's parameters, whereas covariance is required for output members, which correspond to a method's result type. With this distinction, a purely object-oriented calculus can encode functional-style abstraction with the same safety guarantees.

5.3 Dependencies

Like other members, type members are conceptually selected on values. However, for soundness and decidability, the set of values on which an abstract type member may be selected is restricted to *paths* [39]. Roughly, a path is an immutable value. Formally, type members are selected on types, which may embed paths by way of singleton types, lightweight dependent types. Abstract type members may only be selected on singleton types. Thus, even though we distinguish three levels (values, types, and kinds), they are not strictly separated. Scala has always supported path-dependent types, and now kinds may also depend on types.

It is important to note that these dependencies are quite restricted. Types may only depend on paths, immutable values for which a simple — statically decidable — notion of equality is defined. This design seems like a good trade-off between a fully dependently typed languages (such as Cayenne [4], or Epigram [31]) and a language that maintains a strict phase separation (such as Haskell [23] and Ωmega). Interestingly, singleton types are considered an important pattern in Ωmega [43].

6 Type Constructors and Variance

Another facet of the interaction between subtyping and type constructors is seen in Scala's support for definition-site variance annotations [17]. Essentially, variance anno-

tations 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 our model of kinds as did bounded type parameters: our 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 we should track it in $\rightarrow$

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 23 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 we allowed a type constructor that is invariant in its first type parameter, to 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 23 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 Related Work

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 [20,42,7] have served as the basis for many programming languages. The most notable example is Haskell [23], which haves supported higher-kinded types for over 15 years [22].

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 [47]. However, it is not (yet) possible to abstract over class contexts [24,27,29,14]. In our setting, this corresponds to abstracting over a type that is used as a bound, as discussed in Section 4.

The interaction between higher-kinded types and subtyping is a well-studied subject [11,41,16]. As far as we know, none of these approaches combine bounded type

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

class Cell[A] extends Seq[A, Cell] { // (only) compile-time 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()  // run-time error if compile-time error is ignored
```

Listing 23. Example of unsoundness if higher-order variance annotations are not enforced.

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 [10], which corresponds to our bounds-tracking kind `*(L, U)`.

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 [3] partly inspired the design of our syntax. Other than that, we are not aware of other contemporary object-oriented languages with a similar set of features. 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" aims to alleviate this [21].

8 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, genericity's first-order nature makes it self-defeating: 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 experimented with. Embedded domain specific languages (DSL's) [13] are another promising application area of type constructor polymorphism. We are currently applying these ideas to our parser combinator library, a DSL for writing EBNF grammars in Scala [34]. Independently, Ostermann et al. [40] are investigating similar applications, which critically rely on type constructor polymorphism.

9 Acknowledgements

The authors would like to thank Dave Clarke, Marko van Dooren, Burak Emir, Erik Ernst, Bart Jacobs, Jan Smans, and Alexander Spoon for their insightful comments and interesting discussions. We also gratefully acknowledge the Scala community for providing a fertile testbed for this research.

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. L. Augustsson. Cayenne - a language with dependent types. In *Advanced Functional Programming*, pages 240–267, 1998.
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. L. Cardelli. Structural subtyping and the notion of power type. In *POPL*, pages 70–79, 1988.
11. 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.
12. L. Cardelli, S. Martini, J. C. Mitchell, and A. Scedrov. An extension of System F with subtyping. *Inf. Comput.*, 109(1/2):4–56, 1994.
13. 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.
14. 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`.
15. 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.
16. A. B. Compagnoni and H. Goguen. Typed operational semantics for higher-order subtyping. *Inf. Comput.*, 184(2):242–297, 2003.
17. 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.
18. 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.
19. E. Ernst. Family polymorphism. In J. L. Knudsen, editor, *ECOOP*, volume 2072 of *Lecture Notes in Computer Science*, pages 303–326. Springer, 2001.
20. J. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These d'Etat, Paris VII, 1972.
21. 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.
22. 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.
23. 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.
24. J. Hughes. Restricted datatypes in Haskell. Technical Report UU-CS-1999-28, Department of Information and Computing Sciences, Utrecht University, 1999.
25. G. Hutton and E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
26. 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.
27. M. P. Jones. constructor classes & "set" monad?, 1994. `http://groups.google.com/group/comp.lang.functional/msg/e10290b2511c65f0`.
28. M. P. Jones. A system of constructor classes: Overloading and implicit higher-order polymorphism. *J. Funct. Program.*, 5(1):1–35, 1995.
29. E. Kidd. How to make data.set a monad, 2007. `http://www.randomhacks.net/articles/2007/03/15/data-set-monad-haskell-macros`.
30. 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.

31. C. McBride. Epigram: Practical programming with dependent types. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming*, volume 3622 of *Lecture Notes in Computer Science*, pages 130–170. Springer, 2004.
32. 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.
33. 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.
34. A. Moors, F. Piessens, and M. Odersky. Parser combinators in Scala. Technical Report CW491, Department of Computer Science, K.U. Leuven, 2007. Under preparation. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW491.abs.html`.
35. 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/`.
36. M. Odersky. Poor man's type classes, July 2006. Talk at IFIP WG 2.8, Boston.
37. M. Odersky. *The Scala Language Specification, Version 2.6*. EPFL, Nov. 2007. `http://www.scala-lang.org/docu/files/ScalaReference.pdf`.
38. 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.
39. 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.
40. K. Ostermann and C. Hofer, 2007. Private communication.
41. B. C. Pierce and M. Steffen. Higher-order subtyping. *Theor. Comput. Sci.*, 176(1-2):235–282, 1997.
42. 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.
43. T. Sheard. Type-level computation using narrowing in Ωmega. In *PLPV*, 2006.
44. 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.
45. P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.
46. 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.
47. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.
48. 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.