

# Towards Equal Rights for Higher-kinded Types

Adriaan Moors<sup>1\*</sup>, Frank Piessens<sup>1</sup>, and Martin Odersky<sup>2</sup>

<sup>1</sup> K.U. Leuven

{adriaan, frank}@cs.kuleuven.be

<sup>2</sup> EPFL

martin.odersky@epfl.ch

**Abstract.** Generics are a very popular feature of contemporary OO languages, such as Java, C# or Scala. Their support for genericity is lacking, however. The problem is that they only support abstracting over proper types, and not over generic types. This limitation makes it impossible to, e.g., define a precise interface for `Iterable`, a core abstraction in Scala’s collection API. We implemented “type constructor polymorphism” in Scala 2.5, which solves this problem at the root, thus greatly reducing the duplication of type signatures and code.

## 1 Introduction

Object-oriented languages such as Java, C#, and (until now) Scala offer only limited support for genericity, in that generic types are not considered first-class types. In this section, we illustrate why this limitation is problematic using a core abstraction from Scala’s collections API and sketch the solution we implemented in Scala 2.5.

The following sections provides a more detailed discussion of type constructor polymorphism and elaborate on interesting applications. Finally, the section on related work discusses how type constructor polymorphism has been known for several decades in research on functional programming (FP) languages. Industrial-strength FP languages, such as Haskell, have supported it for at least 15 years.

### 1.1 The trouble with `Iterable`

The `Iterable` interface (an extract is shown below) declares the operations that underlie Scala’s for-comprehensions. `map` takes a function from `E1` to `NewE1` and applies it to every element of the current collection to produce a new collection of `NewE1`’s. `flatMap` generalises this behaviour in that, for every element of type `E1`, the user-supplied function may produce a collection of `NewE1`’s, instead of just a single element. The produced elements will all be merged into one collection. Finally, `filter` examines every element in the current collection and returns a new one containing only those elements that matched the user-supplied predicate `p`.

---

\* 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 the third author’s lab.

```

trait Iterable[El] {
  def map[NewEl] (f: El ⇒ NewEl): Iterable[NewEl]
  def flatMap[NewEl] (f: El ⇒ Iterable[NewEl]): Iterable[NewEl]
  def filter(p: El ⇒ Boolean): Iterable[El]
}

```

The signatures of `Iterable`’s methods precisely track the type of the *elements* in the containers they manipulate. However, little is known (or enforced) about the *container* itself. A container of elements of type `T` produced by one of these methods must simply be a subtype of `Iterable[T]`. Actual implementations of these methods allow for stricter bounds on their return types: in a subclass of `Iterable`, the precise type of the produced container is known, and this information should be exposed to the clients of these methods (to a certain extent).

The obvious<sup>3</sup> way to improve `Iterable` so that subclasses can precisely specify which type of container they produce, is to abstract over that type. However, we run into problems when trying to define a type parameter `Container` that could later be instantiated in a subclass. We must be able to apply different type arguments to `Container` (`NewEl` and `El`), but this is not allowed:

```

trait Iterable[El, Container] {
  // error: Container does not take type parameters
  def map[NewEl] (f: El ⇒ NewEl): Container[NewEl]
  def flatMap[NewEl] (f: El ⇒ Container[NewEl]): Container[NewEl]
  def filter(p: El ⇒ Boolean): Container[El]
}

```

Our extension adds support for type parameters that take type parameters themselves. We will elaborate on this in the rest of the paper. First, we show the ad-hoc solution currently employed in the Scala libraries.

Currently, every subclass of `Iterable` refines the result type of the relevant methods individually, scattering this part of `Iterable`’s contract over the class hierarchy. Furthermore, this kind of change is only allowed for result types, as they are in a co-variant position. More complicated designs quickly transcend these limitations.

As an example of this redundancy, consider the `List` subclass:

```

class List[El] extends Iterable[El] {
  def map[NewEl] (f: El ⇒ NewEl): List[NewEl]
  def flatMap[NewEl] (f: El ⇒ Iterable[NewEl]): List[NewEl]
  def filter(p: El ⇒ Boolean): List[El]
}

```

We must refine every single method inherited from the `Iterable` interface to denote the fact that these methods actually produce the same container as the one they were applied to. Worse, almost the same code has to be written over and over again in the subclasses of `Iterable`. Our extension not only allows these methods to receive

<sup>3</sup> Bruce’s `MyType` [4] or Eiffel’s type anchors [12] are not viable alternatives, as the result types are not necessarily the same as the type of the enclosing class: `Iterable[El]`’s methods return `Iterable[El]`’s as well as `Iterable[NewEl]`’s. Furthermore, our approach fully supports programming against interfaces, whereas these alternatives do not.

a more precise signature, it also makes it possible to implement them at the level of `Iterable`, factoring out only the small part that varies over the subclasses.

If `List` did not refine the result type of these methods to reflect its implementation, the following piece of client code would not type check (even though the *run-time* type of `ys` would indeed be `List[String]`):

```
val xs: List[Int] = List(1, 2, 3)
val ys = xs.map(_.toString) // ys has type Iterable[String]

assert(xs.length == ys.length) // error: length is not a member of
    Iterable[String]
```

At the very least, transforming a finite collection should be constrained to result in a finite collection (i.e., one that has a `length`).

The next few paragraphs introduce our extension and illustrate how it solves these problems more systematically.

## 1.2 Type constructor polymorphism

In the above, `List` and `Iterable` are *type constructors*. `List` takes one type parameter, and applying it to a type argument `Int` yields the *proper type* `List[Int]`. We put the emphasis on whether a type expects type arguments. Thus, a type that expects type arguments is a “type constructor” and a type that does not expect type arguments is a “proper type”, whether it be a type constructor applied to type arguments, such as `List[Int]` or simply `Int`.

From now on, type constructors have the same first-class status as proper types, and we shall refer to both as “types”. “Parametric polymorphism” is the language mechanism that abstracts over types using type parameters. When disambiguation is necessary, “type constructor polymorphism” emphasises type *constructors* are abstracted over. We avoid the overloaded term “genericity”.

In Java, type constructors are called “generic types”. A generic type that has been applied to actual arguments, is (confusingly) referred to as a *parameterised type* – we simply consider this a proper type, as it does not expect any (further) type arguments. In a sense, type constructor polymorphism may be thought of as a safe way of dealing with “raw types”.

Type constructors are also known as “higher-kinded types” in functional programming languages, which have been putting them to good use for over a decade. Section 4 goes into more detail on this, the rest of the paper does not require any knowledge of functional programming. We do assume some familiarity with Scala [33].

## 1.3 A better iterable

With type constructor polymorphism, we can refine the definition of `Iterable` as shown below. Now, a subclass, such as `List`, can instantiate a single abstract type *constructor* parameter (`Container`, which takes one type parameter) to accomplish the refinement we were after.

```

trait Iterable[El, Container[_]] {
  def map[NewEl] (f: El ⇒ NewEl): Container[NewEl]
  def flatMap[NewEl] (f: El ⇒ Iterable[NewEl]): Container[NewEl]
  def filter(p: El ⇒ Boolean): Container[El]
}

abstract class List[El] extends Iterable[El, List]

```

Section 3 discusses a more detailed implementation of `Iterable`.

## 2 Informal Description

In this section we’ll elaborate on the impact of type constructor polymorphism on the language. First, the level of kinds is defined so that it integrates well with Scala’s existing features. We gradually build up the kind system to illustrate the complications that arise. Finally, well before version 2.5, Scala already allowed type constructor polymorphism to be encoded to a certain extent. We explain the encoding and contrast it with the direct support.

### 2.1 Kinds

If we are to give type constructors the same status as proper types, we must not gloss over of what makes them different: (most) proper types have instances, whereas a type constructor must first be applied to correct type arguments: this then yields a proper type, which in turn may be used to construct values. This alludes to a nice analogy: type constructors are the type-level equivalent of value-level functions. We will elaborate on this in section 2.5.

One may wonder why we should bother at all with types that do not immediately produce objects. However, if we did not care for those types, we should immediately abolish abstract classes and — on a different level — first-class functions, since the former can’t be instantiated, and (almost) the only thing we can do with a function is to apply it to actual arguments, so that we get the object we are interested in. These analogies are meant to build intuition, they are certainly not the final word on the utility of type constructors.

To 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). Objects are classified by types, which reside in the next level. Finally, types are classified by kinds. As higher levels imply a higher level of abstraction, the number of “entities” in a level becomes more constrained as we go up in the hierarchy: the number of objects isn’t known until run time, types must be known at compile time, and the number of essentially different kinds is fixed by the language specification.

Unlike types, kinds are purely structural: they simply reflect the kinds of the type parameters a type expects. Since proper types all take the same number of type parameters (i.e., none), they are classified by the same kind, which is called `*`. To classify type constructors, we need exactly one more kind. Or rather, a kind *constructor*  $\text{From} \rightarrow \text{To}$ , which abstracts over the kinds `From` and `To`. `From` is the kind of the expected type

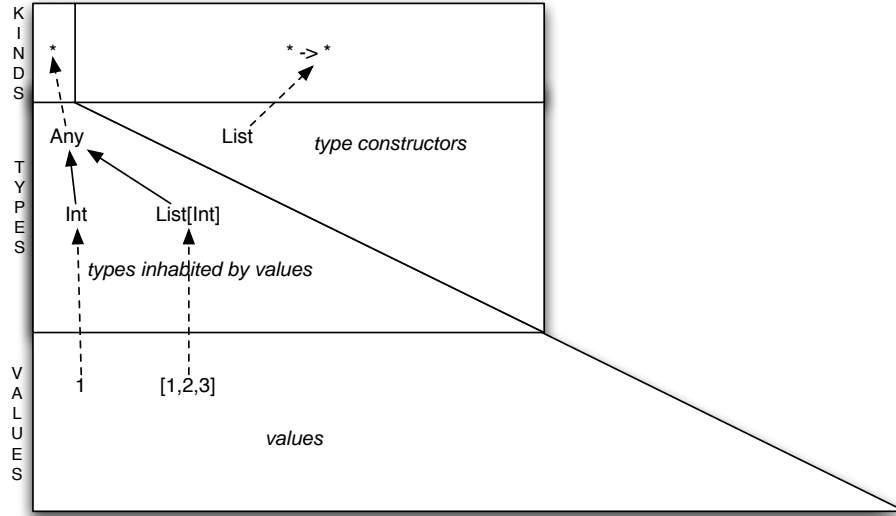


Fig. 1. Diagram of levels

argument and  $T_0$  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  $* \rightarrow *$ , 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`.

## 2.2 Kinds and subtyping

Subtyping plays an important role in building abstractions. As such, it interacts with type constructors in interesting ways. For example, what's the kind of `Carnivore` in `class Carnivore[FoodType <: Meat]`? Simply classifying it as  $* \rightarrow *$  clearly ignores the bound on `FoodType`. We will see later why this distinction is important. First, we incorporate bounds in our notion of kinds.

Before we set out to improve our kind system, let us take stock of the system by defining it explicitly. Since kinds are not user-definable, we invent some syntax based on the existing mechanisms for defining classes. Defining a kind looks just like defining a type, except for the more fitting `kind` keyword.

```
kind *
kind  $\rightarrow$  [From, To]
```

The essential difference between the definitions `kind *` and `class *` is of course that  $*$ 's instances are types – the proper types, to be exact. The kind  $\rightarrow$  takes two kind parameters, `From` and `To`. `From  $\rightarrow$  To`, which is infix notation for  `$\rightarrow$  [From, To]`, classifies the type constructors that take a type argument of kind `From` and yield a type

of kind `To`. This model does not contain any references to types, so kinds cannot denote type bounds such as `<: Meat`.

Instead of re-using Scala’s class system, perhaps a more standard way of describing the level of kinds is using BNF. Note that kinds are not defined explicitly in Scala, so these syntactic constructs do not appear in Scala programs. That said, the syntax of kinds is described as:

```
Kind ::= '*' | Kind '→' Kind
```

To model bounded types as types that are classified by a certain kind, we enrich the definition of `*`, so that it includes the lower and upper bounds that must be satisfied by the types it classifies:

```
kind *(lower: *, upper: *)
```

A type `T` is classified by `*(L, U)` if it is a supertype of `L` and a subtype of `U`.

Syntactically, the `Kind` production thus becomes (assuming the appropriate `Type` production):

```
Kind ::= '*' '(' Type ',' Type ')' | Kind '→' Kind
```

Note that we maintain the convention that (statically known) parameters that are in the same level as the “function” they are applied to, are written in `[...]`, while parameters from a lower level are enclosed in `(...)`. Values are always surrounded by `(...)`, to emphasise that they are not statically known.

This improvement neatly incorporates bounded types into our three-level model of classification: `FoodType <: Meat` is translated into `FoodType : *(Nothing, Meat)`. (Nothing is the default lower bound.)

Now we can properly classify `Carnivore` as `→[*(Nothing, Meat), *(Nothing, Any)]`, or, using more convenient syntax: `*(Meat) → *`. We’ll keep using `*` as shorthand for `*(Nothing, Any)`, and, since we predominantly use upper bounds, `*(T)` abbreviates `*(Nothing, T)`.

## 2.3 Kinds and variance

Another facet of the interaction between subtyping and type constructors is seen in Scala’s support for definition-site variance annotations [14]. Essentially, 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 these type arguments must be identical for the constructed types to be comparable.

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. We won’t go

into much detail here, but the same strategy as for including bounds into  $*$  can be applied here, except that variance is a property of type *constructors*, so we track it in  $\rightarrow$ :

```
sealed kind Variance // an enumeration with three elements
  abstract class Covariance : Variance
  abstract class Contravariance : Variance
  abstract class Invariance : Variance
```

```
kind  $\rightarrow$ [From, To] (variance: Variance)
```

Thus, `List :  $\rightarrow$ [*, *] (Covariance)`, or abbreviated: `List : *  $\xrightarrow{+}$  *`.

The syntactic description is extended correspondingly:

```
Kind ::= '*' '(' Type ',' Type ')' | Kind VarianceArrow Kind
VarianceArrow ::= ' $\xrightarrow{+}$ ' | ' $\xrightarrow{-}$ ' | ' $\rightarrow$ '
```

## 2.4 Syntax

Although kinds play an important role on the conceptual level, they do not manifest themselves syntactically in Scala. In this respect, they are much like method types, which are only expressed indirectly by a method's signature.

The only syntactic change required to support type constructor polymorphism, is that abstract type members and type parameters may now declare type parameters. Before, the signature of an abstract type (the collective term for type parameters and abstract type members) was just an identifier followed by bounds. Now, the identifier may additionally be followed by a list of formal type parameters.

For example, the type parameter `Container[x]` is declared to take one type parameter. Thus, `Container` is of kind  $* \rightarrow *$ , and `Container[Int]` is of kind  $*$ , as discussed earlier. When `x` is not used, it may be replaced by a `_` wild-card: `Container[_]`. Note that `x` is in scope in the immediately enclosing list of parameters as well as in `Container`'s bounds. A more complicated definition, such as `Container[x <: Bound[x]] <: Iterable[x, Bound, Container]` illustrates this. The next section extends our notion of kinds to deal with this final complication.

## 2.5 F-bounds

There's one more feature of bounded types that complicates our definition of kinds [6]. We must extend our model to deal with the possibility that a type's bounds mention the type itself, such as in `class OrderedSet[T <: Ordered[T]]`. Our current notation provides no way to fill in the `??` in `OrderedSet : *(Ordered[??])  $\rightarrow$  *`. Here, the `??`'s refer to `T`, the type argument expected by `OrderedSet`.

To understand this better, let's go down a level and see how this works for value-level functions. As far as dependencies are concerned, the kind of `OrderedSet` is very similar to the function type that corresponds to the method `def id(x: Any): x.type`. A valid function type would look like `Any  $\Rightarrow$  ??.type`, except that the `??` cannot be filled in, as `id`'s argument cannot be referenced in the type. Since Scala

models functions simply as objects with `apply` methods, we can however express this type as:

```
class IdFunction {
  def apply(x: Any): x.type
}
```

Comparing this to the general definition of a function of one argument, makes clear that the type parameter `To` must be able to refer to `x`:

```
// From ⇒ To is syntactic sugar for Function1[From, To]
class Function1[From, To] {    // From, To are types
  def apply(x: From): To      // x is a value
}
```

Thus, if we take this one level higher, and rename  $\rightarrow$  to `TypeFunction1` to emphasise the similarities, we get (in pseudo-Scala):

```
kind TypeFunction1[From, To] { // From, To are kinds
  type apply[tp: From]: To     // tp is a type
}
```

Or, concretely for `OrderedSet`:

```
kind OrderedSetFunction {
  type apply[tp : *(Nothing, Ordered[tp])]: *
}
```

So, in this case, the `From` kind parameter has to be able to reference the type it classifies (`tp`).

We will use the notation  $x@K \rightarrow K'$  to succinctly write down these dependent function kinds. The  $x$  may be used to reference the type classified by  $K$ . A dependent function type is written similarly:  $x@T \Rightarrow T'$ . This allows us to classify `OrderedSet` as  $T@*(Ordered[T]) \rightarrow *$ , or even better, as  $T@*(Ordered[T]) \rightarrow *(OrderedSet[T])$ .

Finally, the full syntax for kinds can be described as:

```
Kind ::= ' * ' ' ( ' Type ' , ' Type ' ) ' | KindArgs VarianceArrow Kind
KindArgs ::= Kind | ' [ ' [Id '@' ] Kind ( ' , ' [Id '@' ] Kind ) * ' ] '
```

```
VarianceArrow ::= '  $\xrightarrow{+}$  ' | '  $\xrightarrow{-}$  ' | '  $\rightarrow$  '
```

## 2.6 Why kinds track bounds

Unfortunately, the previous definition of `Iterable` precludes reusing it for collections that require their elements to be bounded, such as `OrderedSet`:

```
trait OrderedSet[T <: Ordered[T]]
  extends Iterable[T, OrderedSet] // Incorrect!
```

To see why this is problematic, consider the second type argument that `OrderedSet` passes to `Iterable`. This type, `OrderedSet`, is classified by the kind  $T@*(Ordered[T]) \rightarrow *$ . In words, it is a type function that expects a type *that supports ordering* and

returns some proper type. However, `Iterable`’s second type parameter — `Container` — is expected to be of kind  $\ast \rightarrow \ast$ , i.e., it should be a type function that accepts *any* type. Clearly, `OrderedSet` does not respect this contract: it only deals with types  $T$  that are a subtype of  $T <: \text{Ordered}[T]$ .

On the value level, this would correspond to supplying a function of type, say, `String  $\Rightarrow$  Any` when a function of type `Any  $\Rightarrow$  Any` was expected: the latter may be applied to any argument, whereas the former only accepts `String`’s. This is why `String  $\Rightarrow$  Any` is not a *subtype* of `Any  $\Rightarrow$  Any`. Correspondingly, we say that  $T @ \ast ( \text{Ordered}[T] ) \rightarrow \ast$  is not a *subkind* of  $\ast \rightarrow \ast$ .

Luckily, these bounds can be accommodated from the start. This version of `Iterable` abstracts over the upper bound of its elements:

```
trait Iterable[El <: Bound[El], Bound[_], Container[T <: Bound[T]]] {
  def map[NewEl <: Bound[NewEl]](f: El  $\Rightarrow$  NewEl): Container[NewEl]
  def flatMap[NewEl <: Bound[NewEl], NewContainer[T <: Bound[T]]](
    f: El  $\Rightarrow$  Iterable[NewEl, Bound, NewContainer]): Container[NewEl]
  def filter(p: El  $\Rightarrow$  Boolean): Container[El]
}

trait OrderedSet[T <: Ordered[T]]
  extends Iterable[T, Ordered, OrderedSet]
```

Note that this version strictly generalises the previous one, as we may instantiate `Bound` to the trivial upper-bound, `Any`. Note that — for convenience — `Any` is “kind-overloaded”: it is the top type for all well-kinded types. (Likewise, `Nothing` is a subtype of every well-kinded type.)

## 2.7 Subkinding

In this section we make the notion of *subkinding* more precise. The similarities with subtyping are striking.

Subkinding for the kinds that classify proper types, simply corresponds to subtyping the corresponding bounds. Type bounds with lower bound  $S$  and upper bound  $T$  are subsumed by type bounds with a “lower” lower bound and a “greater” upper bound:

$$\frac{\begin{array}{l} \Gamma \vdash S' <: S \\ \Gamma \vdash T <: T' \end{array}}{\Gamma \vdash \ast(S, T) <: \ast(S', T')} \quad \text{K\_SUB\_STAR}$$

For our dependent function kinds,  $[x_1@K_1, \dots, x_n@K_n] \rightarrow K$ , matters become slightly more complicated. Such a function may be passed whenever a function that accepted “fewer” arguments and returned “worse” results was expected. More precisely,  $[x_1@K_1, \dots, x_n@K_n] \rightarrow K$  is a sub-kind of  $[x_1@K'_1, \dots, x_n@K'_n] \rightarrow K'$  if their argument kinds vary contravariantly ( $K'_i <: K_i$ ), where both  $K_i$  and  $K'_i$  may contain the free variables  $x_i$  that denote the type they classify, and if, assuming the “best” kind  $K'_i$  for that type argument  $x_i$ , the result kinds behave covariantly.

$$\frac{\begin{array}{c} \Gamma \vdash K'_i <: K_i \\ \Gamma, X_i : K'_i \vdash K <: K' \end{array}}{\Gamma \vdash [X_1 @ K_1, \dots, X_n @ K_n] \rightarrow K <: [X_1 @ K'_1, \dots, X_n @ K'_n] \rightarrow K'} \quad \text{K\_SUB\_ARR}$$

Note that — modulo the syntax — this rule is identical to the rule for *subtyping* dependent function types [2].

## 2.8 Kind assignment

The rules that define the well-formedness of types in a single-kinded language, correspond to the rules that assign a kind  $*$  to a type. Parametric polymorphism essentially adds two more forms of types that may be well-kinded: type abstractions and type applications.

A type that abstracts over types  $T_1$  to  $T_n$  to yield a type  $T$ , which may contain references to the  $T_i$ , receives the kind  $T_1 @ K_1 \rightarrow \dots \rightarrow T_n @ K_n \rightarrow K$  if  $T$  has the kind  $K$  under the assumption that the  $T_i$  have kind  $K_i$ . Applying such a type abstraction to  $n$  type arguments  $S_i$  with kind  $K_i$  again results in a type with kind  $K$  (with a suitable simultaneous substitution of the type parameters).

Finally, saying that a type has “a” kind is slightly misleading, as well-kinded types have many kinds. This is due to our — somewhat non-standard — approach of tracking bounds in the  $*$ -kind, so that all of the following hold:

```
Int : *(Nothing, Any)
Int : *(Nothing, Int)
Int : *(Int, Int)
```

At this point, it is unclear what the impact of this feature is on the meta-theory of the underlying calculus. Intuitively, we consider it an elegant way of keeping the levels of types and kinds as similar as possible. “is-of-type”-constraints on values correspond nicely to “is-subtype-of” constraints on types, therefore we model both as classification by a meta-entity.

Furthermore, the “singleton” kind  $*(Int, Int)$  seems an interesting way to express “exact types” — types that are not subject to subsumption. We have yet to explore their applications in detail, but the interaction with virtual classes seems promising.

Until now, we have only used the functional style of abstraction and application, that is, using type parameters. 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 limited support for type constructor polymorphism before our extension.

## 2.9 Encoding type constructors using abstract type members

With its second release, Scala added support for abstract type member refinement, which allowed for an encoding of higher-kinded types using abstract type members. Type application could be accomplished through abstract type member refinement.

While this encoding theoretically fully supports type constructor polymorphism, it requires a global rewrite of all parameterised types involved. Type parameters must be replaced by abstract type members, and type application is achieved by overriding abstract type members with concrete ones.

Besides the non-local character of the encoding, it does not detect erroneous type applications as early as possible. More precisely, a well-typed program may still contain ill-formed (encodings of) type applications, which result in empty (uninhabited) types. Consider the encoding of our running example:

```
trait TypeConstructor1 { type A }

trait Iterable extends TypeConstructor1 {
  type Container <: TypeConstructor1

  def map[B] (f: A => B): Container{type A = B}
  def flatMap[B] (f: A => Iterable{type A = B}): Container{type A = B}
  def filter(p: A => Boolean): Container{type A = Iterable.this.A}
}

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

Now, what happens when we encode the erroneous version of `OrderedSet`? The compiler rejects the following program, as we discussed in section 2.6:

```
trait OrderedSet[T <: Ordered[T]]
  extends Iterable[T, OrderedSet] // Incorrect, compiler error.
```

However, the encoded version below is accepted without a warning or error.

```
trait OrderedSet extends Iterable {
  type A <: Ordered[A]
  type Container = OrderedSet // Incorrect, but no error reported!
}
```

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.

A similar situation arises with abstract classes. Translating the leniency with type members to this context would mean that any class would be allowed to be abstract implicitly, so that, until an attempt is made to instantiate this class, the oversight might go unnoticed. Based on this precedent, erroneous type applications should be detected as early as possible too.

In fact, we consider it a “kind unsoundness” that well-kinded types may contain type applications that “go wrong”. The problem lies in the fact that the type function `OrderedSet` is allowed to strengthen its type argument `A` (encoded as the abstract type

member  $A$ ), while still being considered a subtype of `TypeConstructor1` (i.e., it is considered of kind  $* \rightarrow *$ ). Section 2.6 discussed in more detail why  $T@*(Ordered[T]) \rightarrow *$  (the kind of `OrderedSet`) should be distinguished from  $* \rightarrow *$ .

This discrepancy has its roots in the  $\nu$ Obj calculus [34], which allows abstract type members to be refined *covariantly*, thus `OrderedSet <: TypeConstructor1`. This is at odds with our definition of subkinding and the assumption that types can only be subtypes when their kinds are subkinds: here, `OrderedSet <: TypeConstructor1` although it is not the case that  $T@*(Ordered[T]) \rightarrow * <: * \rightarrow *$ .

Related work seems to deviate from  $\nu$ Obj’s design, although making a precise comparison is complicated by the differences in features supported by the various approaches. In the notation of Cardelli [8], the main two types are classified as follows:

```
OrderedSet : ALL[A::POWER[Ordered[A]]] TYPE
TypeConstructor1 : 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 [11]:

```
OrderedSet : Pi A <: Ordered A : *. *
TypeConstructor1 : 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 [9] rule that deals with bounded quantification at the value level (Sub Forall) also requires *contravariance* for the bounds of the quantifier.

## 2.10 Dependencies

Even though we distinguish three levels (values, types, and kinds), they are not strictly separated. Scala has always allowed types to depend on certain values (called “paths”), and now kinds may also depend on types. The former allows for sound late-bound (“virtual” or “abstract”) type members, and the latter is a natural consequence of the interaction between subtyping and kinds.

It is important to note that these dependencies are quite restricted. Types may only depend on immutable values, called paths, 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 [3], Epigram [28], ...) and a language that maintains a strict phase separation (such as Haskell [20] and  $\Omega$ mega). Interestingly, singleton types are considered an important pattern in  $\Omega$ mega [38].

As mentioned earlier, singleton kinds are an artefact of our conceptual model that we have not explored yet. They could be used to support exact types, i.e., types that are not subject to subsumption. These types provide more information about their values.

As such, they may make virtual classes more practical in that they provide more opportunities for detecting that virtual types are comparable. Exact types are not new: in F#, for example, a type (ascription) has to be made “eligible” for subsumption explicitly [39].

### 3 Examples

#### 3.1 Implementing Iterable

In this section we demonstrate how type constructor polymorphism can be leveraged to define a more precise — and at the same time more compact — `Iterable`. To do this, we introduce the `Builder` and `Buildable` abstractions, which may be considered the “duals” of `Iterator` and `Iterable`. An `Iterable` provides a way to abstractly consume the elements of a collection, whereas a `Builder` is used to abstractly produce a collection from its individual elements.

To continue the `Iterator/Builder` analogy on the level of methods: a client “pulls” elements from a collection using `next`, and `+=` “pushes” elements to the collection under construction. Finally, `hasNext` signals that the collection is exhausted, while the constructed collection is retrieved using `finalise`. `using` and `foreach` are obvious convenience methods.

```
trait Builder[Container[_], El] {
  def +=(el: El): Unit
  def finalise(): Container[El]

  def using(op: this.type => Unit): Container[El] = {
    op(this); finalise() }
}

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

  def foreach(op: El => Unit): Unit = while(hasNext) op(next())
}
```

Most importantly, note that `Builder` makes precise which kind of container it produces using the `Container` type constructor parameter. For now, the `+` variance annotation on `Iterator`’s type parameter may safely be ignored.

Before we get to the implementation of `Iterable`, we show its collection-producing counterpart, `Buildable`:

```
trait Buildable[Container[_]] {
  def builder[T]: Builder[Container, T]

  def fromIterator[El, NewEl](elems: Iterator[El])
    (op: Builder[Container, NewEl] => El => Unit): Container[NewEl]
    = builder.using{ b => elems.foreach(op(b)) }
}
```

The method `fromIterator` constructs a `Container[NewEl]` by applying the user-supplied operation `op` to the right `Builder` and to each of the elements that should be in the resulting container. With these fundamental operations, we can reduce `Iterable`'s core methods to their very essence.

To avoid distractions, we first show a slightly simplified implementation, omitting how the `b` of type `Buildable[Container]` is supplied and delaying another generalisation. For every element in the `Iterable`, `filter` passes it to the builder if the user-supplied predicate `p` yields `true` for this element. Similarly, `map` transforms each element using `f` and passes the result to the builder. Finally, `flatMap` invokes the builder for every individual element returned by `f`.

```
def filter(p: El => Boolean) = b.fromIterator(elements){
  builder => el => if(p(el)) builder += el
}
def map[NewEl](f: El => NewEl) = b.fromIterator(elements){
  builder => el => builder += f(el)
}
def flatMap[NewEl](f: El => Iterable[NewEl]) =
  b.fromIterator(elements){ builder => el =>
    f(el).elements.foreach{ el => builder += el }
}
```

The complete implementation, shown in Fig. 1.1, differs only slightly. The `filter/map/flatMap` trio is first implemented more generally, allowing the caller to specify explicitly which type of collection should be used to contain the resulting elements. The traditional methods can be implemented trivially on top of the more general ones. For convenience, `Container` is an abstract type member, which may be thought of as a type parameter that can be referenced by name from outside `Iterable`. Therefore, it need not be made explicit every time the type `Iterable[T]` is written.

The second simplification was the omission of the implicit argument list from the methods, so that we could not explain how the methods knew which instance of `Buildable[Container]` to use. Each of them receives this instance as the implicit argument `b`. If an implicit argument is not specified by the programmer, the compiler automatically supplies the matching implicit value that is in scope (if there is exactly one such value). Thus, it is easy to chose different building strategies without changing the rest of the behaviour of the container.

The listing in Fig. 1.2 and 1.3 show the implicit objects that implement the `Buildable` interface for `List` and `Option`. This style of programming was inspired by Haskell's type classes [40,32,41]. `Buildable` models a type class and the implicit objects declare instances of `Buildable` for `List` and `Option`.

The goal of this section was to show how type constructor polymorphism enables more precise signatures and reduced code duplication. We did not strive for performance. Our implementation certainly is less efficient than the one in the Scala libraries, due to the heavy use of closures to factor out the commonalities between the `filter/map/flatMap` methods. Preliminary micro-benchmarks indicate an overhead between 20% and 30%. We do not go in more detail as these results do not pertain to the use of higher-kinded types.

**Listing 1.1.** A complete implementation of Iterable

```

trait Iterable[+El] {
  type Container[+X] <: Iterable[X]

  def elements: Iterator[El]

  def filterTo[NewContainer[+X]](p: El ⇒ Boolean)
    (implicit b: Buildable[NewContainer]): NewContainer[El]
    = b.fromIterator(elements){ builder ⇒ el ⇒
      if(p(el)) builder += el
    }
  def mapTo[NewEl, NewContainer[_]](f: El ⇒ NewEl)
    (implicit b: Buildable[NewContainer]): NewContainer[NewEl]
    = b.fromIterator(elements){ builder ⇒ el ⇒
      builder += f(el)
    }
  def flatMapTo[NewEl, NewContainer[X]](f: El ⇒ Iterable[NewEl])
    (implicit b: Buildable[NewContainer]): NewContainer[NewEl]
    = b.fromIterator(elements){ builder ⇒ el ⇒
      f(el).elements.foreach{ el ⇒ builder += el }
    }

  def map[NewEl](f: El ⇒ NewEl)
    (implicit b: Buildable[Container]): Container[NewEl]
    = mapTo[NewEl, Container](f)
  def filter(p: El ⇒ Boolean)
    (implicit b: Buildable[Container]): Container[El]
    = filterTo[Container](p)
  def flatMap[NewEl](f: El ⇒ Iterable[NewEl])
    (implicit b: Buildable[Container]): Container[NewEl]
    = flatMapTo[NewEl, Container](f)
}

```

**Listing 1.2.** Supplying the strategy to build a List

```

implicit object ListIsBuildable extends Buildable[List] {
  def builder[El]: Builder[List, El] = new ListBuffer[El] with
    Builder[List, El] {
    def finalise(): List[El] = toList
  }
}

```

**Listing 1.3.** Supplying the strategy to build an Option

```

implicit object OptionIsBuildable extends Buildable[Option] {
  def builder[E1]: Builder[Option, E1] = new Builder[Option, E1] {
    var res: Option[E1] = None()

    def +=(el: E1) = if(res.isEmpty) res = Some(el)
      else throw new UnsupportedOperationException("An Option holds
        _max_1_element")

    def finalise(): Option[E1] = res
  }
}

```

Finally, this improved `Iterable` interface may be used to type-check for-comprehensions without first expanding them to the corresponding method calls. Due to the former lack of a universal `Iterable` interface, for-comprehensions are purely syntactic sugar. Because the program is type checked after expanding the syntactic sugar, error-reporting is complicated. We leave it to future work to remedy this.

**3.2 Type-level Church encodings**

Scala's kinds correspond to the types of the simply-typed lambda calculus [10]. This means that we can express addition on natural numbers on the level of types using a Church Encoding. A natural number is represented by a type function that takes two arguments: the successor function and the zero element. 0 is then encoded as the type function that simply returns its second argument:

```
type _0[s[_], z] = z
```

Subsequent naturals are expressed using repeated application of the supplied successor function:

```

type _1[s[_], z] = s[z]
type _2[s[_], z] = s[s[z]]

```

Given these definitions, addition is a type function that takes two natural numbers as well as the successor function and the zero element. These are then composed so that one of the terms becomes the zero element of the other term:

```

type plus[m[s[_], z], n[s[_], z], s[_], z] = n[s, m[s, z]]

type _3[s[_], z] = plus[_1, _2, s, z]
      // = _2[s, _1[s, z]] = s[s[_1[s, z]]]
      // = s[s[s[z]]]

```

Scala allows type constructors to be written infix, so that, if we fix the types that represent the neutral element and the successor function, we can simply write `_2 + _2`:

```

abstract class Zero
abstract class Succ[a]

type +[m[s[_], z], n[s[_], z]] = plus[m, n, Succ, Zero]
type _4 = _2 + _2

```

Since it is meaningless to instantiate `Zero` and `Succ[a]`, they are declared **abstract**. To check that the type `_2 + _2` actually equals the type that represents 4, we define a helper class `Equals`. If `Equals[A, B]` is well-typed, the types `A` and `B` are equal.

```

case class Equals[A >: B <: B, B]

```

Thus, the equation  $2 + 2 = 4$  is encoded as `Equals[_2 + _2, Succ[Succ[Succ[Zero]]]]`.

## 4 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  $F_\omega$  [15,37,5] have served as the basis for many programming languages. The most notable example is Haskell [20], which has supported higher-kinded types for over 15 years [19].

However, Haskell eschews subtyping. Most of the use-cases for subtyping are subsumed by type classes, a novel mechanism to handle overloading systematically [40]. Still, it does not seem possible to abstract over class contexts [23,26]. In our setting, this corresponds to abstracting over a type that is used as a bound, as discussed in section 2.6.

The interaction between higher-kinded types and subtyping is a well-studied subject [8,36,11]. 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 [7], which corresponds to our bounds-tracking kind  $\ast (L, U)$ .

$\Omega$ mega [38] is a Haskell-based language that (most notably) supports user-defined kinds and type-level computation. To a certain extent, it seems possible to encode the first mechanism using sealed hierarchies of abstract classes in Scala. Scala’s singleton types are a good match for the Singleton pattern, which — according to Sheard — is an important concept in Omega. It may be possible to encode a limited form of type-level computation using Scala’s implicits. However, dedicated support is clearly needed for this feature to be powerful enough. Part of our ongoing work is geared towards bringing the essence of Omega’s power to Scala. The main goal of this effort is to realise an extensible type system that can be used for program verification.

Finally, it seems type constructor polymorphism has only recently started to trickle down to object-oriented languages. Cremet and Altherr’s work on extending Featherweight Generic Java with higher-kinded types [1] partly inspired the design of our syntax. Other than that, we are not aware of other contemporary object-oriented languages with similar features. C++’s template mechanism is closely related. Templates are very flexible, but this comes at a (steep) price: they can only be type-checked after they have been expanded. Recent work on “concepts” alleviates this [16].

**Listing 1.4.** Abstracting over type constructors

```

class Collection[T]
class List[T] extends Collection[T]

abstract class Builder[SomeCollection[X] <: Collection[X]] {
  def build[T]: SomeCollection[T]
}

```

## 5 Conclusion

Although object-oriented languages have adopted “parametric polymorphism” from functional programming (FP) languages as “genericity”, they have not gone all the way yet by also supporting abstracting over generic types. These types are called “higher-kinded types” in FP. We call them “type constructors” and we use “type *constructor* polymorphism” to refer to genericity that deals with type constructors and regular types alike. In other words, a type constructor is like a function on the level of types that takes type arguments to yield a type. Type constructor polymorphism is the type-level equivalent of allowing value-level functions to be used as arguments.

The code fragment in Fig. 1.4 illustrates the utility of this generalisation. Consider the generic class `Collection`, which abstracts over the type of its elements by means of a type parameter `T`. Using type constructor polymorphism, we can provide different strategies to build collections of a given type *constructor*. Thus, `Builder` takes a type constructor parameter `SomeCollection`, which itself takes one type parameter. For any type `T`, `SomeCollection[T]` must be a subtype of `Collection[T]`.

A client can use such a builder to build a specific type of collection. For example, a `Builder[List]` denotes a builder for lists. Invoking `build[Int]` on an instance of `Builder[List]` yields a `List[Int]`. Without type constructor polymorphism, this can only be *approximated* using ad-hoc specialisation and duplication of code and types, which is exactly what genericity tries to avoid.

We implemented type constructor polymorphism in Scala 2.5. The integration with subtyping makes it easy to define abstractions that are quite tricky to express even in Haskell. The prime example is our version of `Iterable` (which is much like a monad) that abstracts over the bound on its elements [23,26]. The current implementation has two known limitations: type constructor parameters are not inferred (we suspect this may be undecidable in our setting) and Scala’s view bounds have not yet been lifted to support higher-kinded type parameters. The latter is only a minor inconvenience, as it can easily be encoded.

We showed how type constructor polymorphism reduces the duplication of code and types in the subclasses of `Iterable`, while at the same time providing a richer and more precise interface. A more in-depth exploration of this design space as well as the search for more applications is a topic of ongoing research. A more refined `Iterable` interface is just one of the many applications of type constructor polymorphism.

Two other examples of promising technologies that rely on type constructor polymorphism, are data-type generic programming (DGP) [24,17,18] and parser combina-

tors [27,22,21]. In previous work, we ported techniques for DGP from Haskell to Scala [30,29]. Recently, we developed a library for parser combinators in Scala [31], and we are currently investigating how type constructor polymorphism can be leveraged to provide more powerful abstractions in this context.

Finally, we are working on a more formal treatment of our extension. For the moment, the most promising strategy seems to be to interpret higher-order polymorphism as the infinite family that results from instantiating the type constructor parameters with all the conforming concrete type constructors. Based on earlier meta-theoretic results for Scala without type constructor polymorphism [13], we aim to prove our extension sound by proving this expansion turns a well-typed program in the calculus with type constructor polymorphism into a well-typed program using only traditional parametric polymorphism. This technique of representing polymorphic types as the corresponding families of their instantiations has previously been employed by Kennedy and Syme [25], and Odersky and Läufer [35].

## 6 Acknowledgements

The authors would like to thank Burak Emir, Bart Jacobs, and Marko van Dooren for their insightful comments and interesting discussions. We also gratefully acknowledge the Scala community, and especially Lauri Alanko, for providing a fertile testbed for this research.

## References

1. 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.
2. D. Aspinall and A. B. Compagnoni. Subtyping dependent types. *Theor. Comput. Sci.*, 266(1-2):273–309, 2001.
3. L. Augustsson. Cayenne - a language with dependent types. In *Advanced Functional Programming*, pages 240–267, 1998.
4. K. B. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *POPL*, pages 285–298, 1993.
5. K. B. Bruce, A. R. Meyer, and J. C. Mitchell. The semantics of second-order lambda calculus. *Inf. Comput.*, 85(1):76–134, 1990.
6. 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.
7. L. Cardelli. Structural subtyping and the notion of power type. In *POPL*, pages 70–79, 1988.
8. 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.
9. 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.
10. A. Church. A formulation of the simple theory of types. *J. Symb. Log.*, 5(2):56–68, 1940.
11. A. B. Compagnoni and H. Goguen. Typed operational semantics for higher-order subtyping. *Inf. Comput.*, 184(2):242–297, 2003.

12. W. R. Cook. A proposal for making Eiffel type-safe. In *ECOOP*, pages 57–70, 1989.
13. V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for Scala type checking. In R. Kralovic and P. Urzyczyn, editors, *MFCS*, volume 4162 of *Lecture Notes in Computer Science*, pages 1–23. Springer, 2006.
14. B. Emir, A. Kennedy, C. V. Russo, and D. Yu. Variance and generalized constraints for C<sup>#</sup> generics. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 279–303. Springer, 2006.
15. J. Girard. Interpretation fonctionnelle et elimination des coupures de l’arithmetique d’ordre superieur. These d’Etat, Paris VII, 1972.
16. 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.
17. R. Hinze. Generics for the masses. In C. Okasaki and K. Fisher, editors, *ICFP*, pages 236–243. ACM, 2004.
18. R. Hinze and J. Jeuring. Generic Haskell: Practice and theory. In R. C. Backhouse and J. Gibbons, editors, *Generic Programming*, volume 2793 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2003.
19. 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.
20. 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.
21. G. Hutton and E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.
22. G. Hutton and E. Meijer. Monadic parsing in Haskell. *J. Funct. Program.*, 8(4):437–444, 1998.
23. M. P. Jones. constructor classes & "set" monad?, 1994. Answer on comp.lang.functional <http://groups.google.com/group/comp.lang.functional/msg/e10290b2511c65f0>.
24. S. L. P. Jones and R. Lämmel. Scrap your boilerplate. In A. Ohori, editor, *APLAS*, volume 2895 of *Lecture Notes in Computer Science*, page 357. Springer, 2003.
25. A. Kennedy and D. Syme. Transposing F to C<sup>#</sup>: expressivity of parametric polymorphism in an object-oriented language. *Concurrency - Practice and Experience*, 16(7):707–733, 2004.
26. E. Kidd. How to make data.set a monad, 2007. Blog post at <http://www.randomhacks.net/articles/2007/03/15/data-set-monad-haskell-macros>.
27. 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.
28. 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.
29. A. Moors. Code-follows-type programming in Scala. Manuscript available from [http://www.cs.kuleuven.be/~adriaan/?q=cft\\_intro](http://www.cs.kuleuven.be/~adriaan/?q=cft_intro), 2007.
30. A. Moors, F. Piessens, and W. Joosen. An object-oriented approach to datatype-generic programming. In *Proceedings of the Workshop on Generic Programming*, 2006.
31. 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>.

32. M. Odersky. Poor man's type classes, July 2006. Talk given at the IFIP WG 2.8 working group, Boston.
33. 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.
34. 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.
35. M. Odersky and K. Läufer. Putting type annotations to work. In *POPL*, pages 54–67, 1996.
36. B. C. Pierce and M. Steffen. Higher-order subtyping. *Theor. Comput. Sci.*, 176(1-2):235–282, 1997.
37. 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.
38. T. Sheard. Type-level computation using narrowing in  $\Omega$ mega. In *PLPV*, 2006.
39. D. Syme, et al. The F# programming language informal specification. <http://research.microsoft.com/fsharp/manual/lexyacc.aspx>, 2007.
40. P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.
41. S. Wehr, R. Lämmel, and P. Thiemann. JavaGI: Generalized interfaces for Java. In *ECOOP 2007, Proceedings*, LNCS. Springer-Verlag, July 2007. 25 pages; To appear.