Typesafe Extensible Functional Objects

Motivation

Scala does not allow mixing in traits into *existing objects*.

Scala supports static composition of traits. It does not allow to extend the interface of already constructed objects. This limitation is common in the field of statically typed, class-based programming languages.
Dynamic specialization of objects enables new aspects of *modularization*, *dynamic adaption* to the computational context as well as *incremental specification* of objects. Examples for dynamic object specialization include:

- adding methods for printing and tracing in order to facilitate debugging,
- creation of mock objects in test-driven-development,
- incremental construction of objects performed by modularized builders and
- annotating objects with additional information, acquired after object creation.

Thus augmenting objects after their construction appears as an attractive language feature.

Approach

The Scala library ***obj*.extend** does allow dynamic specialization of objects.

Building on a coalgebraic encoding [Reichel, 1995, Jacobs, 1995], objects are represented by *terminal coalgebras* – the greatest fixed point $S \xrightarrow{\sim} F[S]$ – over the corresponding *interface endofunctor* F.

Modifying the standard coalgebraic encoding we can model dynamic specialization by composition of coalgebras. The encoding has the following characteristic properties:

- A first class representation of the fixed point as trait Fix
- A definition of the function compose on coalgebras
- The novel observation **extend** which is added to the fixed point and implemented by the composition

The implementation of extension uses that retroactive extension

$\text{unfold}\,(co_1,\, s_1)\ \textbf{extend}\,(co_2,\, s_2)$

can also be expressed as a composition

$\text{unfold}\,(\text{compose}\,(co_1,\, co_2),\, (s_1,\, s_2))$

Static Mixin Composition in Scala

```
val c = new Counter with SkipCounter { var i = 0 }
c.inc
c.skip
```

✗ All traits have to be known at compile time

✗ No object instantiation from type parameters

✗ Objects cannot be extended with trait after construction

Today already possible in Scala: **Static mixin composition**. For instantiation all traits that are mixed into a newly created object have to be statically known.
The required interface here are expressed via a self-type annotation. Self-types allow specifying boundaries on the late bound self-reference without imposing requirements on the actual implementation or the linearization of super-classes.

```
trait Counter {
  private var i: Int = 0
  def get: Int = i
  def inc: Unit = { i += 1 }
}
trait SkipCounter { self: Counter ⇒
  def skip: Unit = { this.inc; this.inc }
}
```

The result type Unit indicates side-effects and a possible change of the internal state.

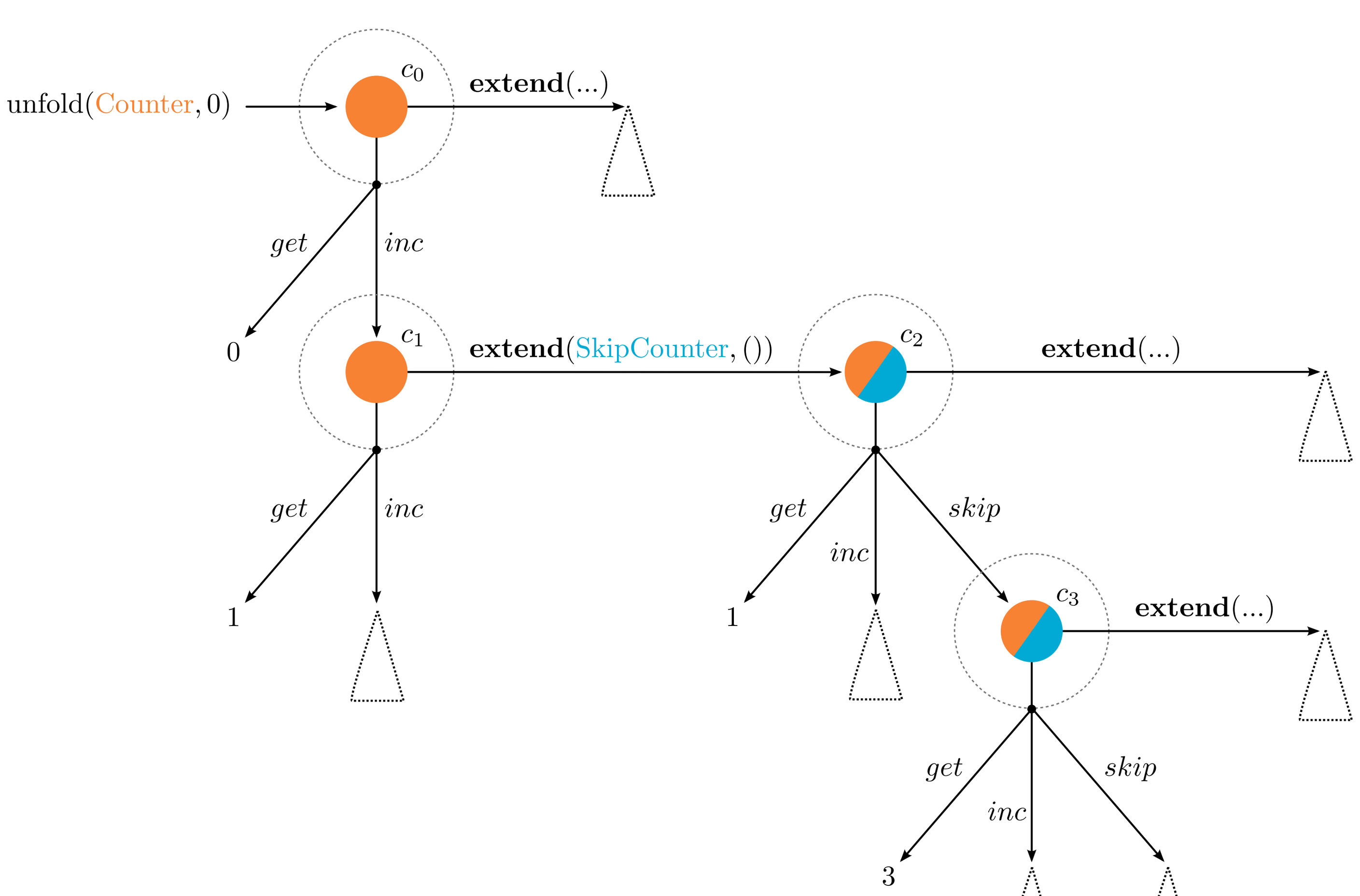
Dynamic Extensibility with *obj*.extend

```
val c0 = unfold (Counter, 0)
val c1 = c0.inc
val c2 = c1 extend (SkipCounter, ())
val c3 = c2.skip
```

```
//                              Self      Provided  State
object Counter extends OpenCoAlg[CounterF, CounterF, Int] {
  def apply[S] (priv: Lens[S, Int]) = self ⇒ state ⇒ new CounterF[S] {
    def get = priv.get (state)
    def inc = priv.set (state, priv.get (state) + 1)
  }
}

trait CounterF[+S] {
  def get: Int
  def inc: S
}

//                                   Self                   Provided  State
object SkipCounter extends OpenCoAlg[CounterF WithF SkipF, SkipF, Unit] {
  def apply[S] (priv: Lens[S, Int]) = self ⇒ state ⇒ new SkipF[S] {
    def skip = self (self (state).inc).inc
  }
}

trait SkipF[+S] {
  def skip: S
}
```

Enabled by the presented encoding: **Dynamic specialization of objects**. Being a functional encoding, the state is threaded through the variables c_i. The corresponding tree of observations for each state i is depicted below. The implementations are defined over the interface endofunctors Counter_F and Skip_F. SkipCounter encodes the self-type annotation via the first type parameter *Self*. The lens *priv* allows accessing and modifying the private state within a contextual frame that forms the overall object state.

Terminal Coalgebras – Extensible Observation Trees



A new observation **extend** allows retroactive composition with another co-algebra.

Coalgebras are blackboxes $S \Rightarrow F[S]$. Where an algebra allows constructing elements of the carrier S given a structure $F[S]$, provided with a state S coalgebras allow making observations structured by $F[S]$.
Unfolding a complete coalgebra to its terminal pendant yields an infinite tree of observations represented by the fixed point:

```
trait Fix[+F[+_]] {
  def out: F[Fix[F]]
  def extend[G[+_], T] (c2: OpenCoAlg[F WithF G, G, T]):
    T ⇒ Fix[F WithF G]
}
```

Usually the tree would only consist of the nodes c_0 and c_1 – however, we embrace the fact that the fixed point is a first class object and add the novel observation **extend**. Calling **extend** with another coalgebra, can be seen as replacing the **extend** observation in the state tree with the result of unfolding the composed coalgebra and thus instantiating the extension point.

Modularity with Open Recursion and Lenses

Self-References are implemented with open recursion, private state using lenses.

Open coalgebras are introduced to allow modular definition of coalgebras that are later composed [Oliveira et al., 2013]. The component coalgebras can have mutual dependencies.

```
trait OpenCoAlg[-Self[+_], +Provided[+_], State] {
  def apply[S] (priv: Lens[S, State]):
    CoAlg[Self, S] ⇒ CoAlg[Provided, S]
}
```

Dependencies can be articulated via the type parameter *Self*, used to describe the late bound coalgebra CoAlg [*Self*, *S*] that can be used to trigger observations implemented by other coalgebras.
Open coalgebras that are self-sufficient are called complete.

```
type CompleteCoAlg[F[+_], S] = OpenCoAlg[F, F, S]
```

Complete open coalgebras can be closed by constructing the fixed point [Cook and Palsberg, 1989]. Hence, the reference *self* is bound late and only determined when the coalgebra is unfolded and the self-reference is closed.

The dotted circle in both illustrations depicts the abstraction boundary that Fix represents. Only when calling *out* the observations described by the interface functor can be made. This allows refining the functor before future calls to *out* are made.
The usage of lenses for accessing private state assures that the full state can be threaded through multiple calls through the self-reference.

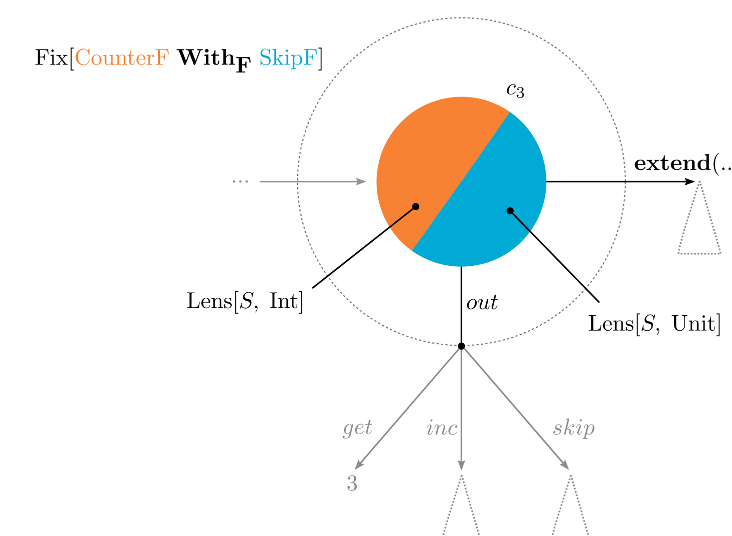


Illustration of the fixed point for the intersection functor Counter_F $\textbf{With}_\textbf{F}$ Skip_F, representing the counter object in state c_3. The components of the private state can be accessed by the coalgebras via lenses.

Composition of Coalgebras

The function compose can be implemented by pointwise application of the coalgebras. Lenses are used to project into the private state. To compose the resulting interface implementations $F[S_1]$ and $G[S_2]$ the *composition type class* $\textbf{With}_\textbf{F}$ is introduced.
An instance of F $\textbf{With}_\textbf{F}$ G is a rank-2 polymorphic evidence, proving that for all types A an object of $F[A]$ and an object of $G[A]$ can be composed to $F[A]$ **with** $G[A]$.

```
trait WithF[F[+_], G[+_]] {
  type Apply[A] = F[A] with G[A]
  def apply[A] (fa: F[A], ga: G[A]): Apply[A]
}
```

Instances are automatically derived by means of reflection and implement the methods of the intersection type by forwarding to *fa* and *ga*.

Conclusions & Future Work

Compared to the decorator pattern, the standard OO-solution for dynamic extensibility, the presented encoding has the advantages that *a*) separately defined extensions can be composed and *b*) open recursion is supported.
Two extensions have been developed that add further functionality to the core as presented in this poster. The first extension allows explicitly calling overridden methods of the extended base, commonly known as "super-calls". The second extension builds on the first and additionally implements *selective open recursion* Aldrich and Donnelly [2004]. Using these extensions a variety of small case studies have been developed that showed the technical feasibility of the approach. However, to be useful in practice future work has to concentrate on a user interface that is syntactically lightweight as well as optimizing the performance of programs written in the encoding.

References

J. Aldrich and K. Donnelly. Selective open recursion: Modular reasoning about components and inheritance. *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 26, 2004.

W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. *SIGPLAN Not.*, 24(10): 433–443, Sept. 1989.

B. Jacobs. *Objects and Classes, Coalgebraically*, pages 83–103. Springer-Verlag, 1995.

B. C. Oliveira, T. V. D. Storm, A. Loh, and W. R. Cook. Feature-Oriented Programming with Object Algebras. In *Proceedings of the European Conference on Object-Oriented Programming*, 2013.

H. Reichel. An approach to object semantics based on terminal co-algebras. *Mathematical Structures in Computer Science*, 5:129–152, 6 1995.

Jonathan Immanuel Brachthäuser
University of Marburg, Germany
jonathan@b-studios.de
`http://files.b-studios.de/icfp2014-poster.pdf`
`http://github.com/b-studios/MixinComposition`

Philipps Universität Marburg