Typesafe **Extensible** Functional Objects

Jonathan Immanuel Brachthäuser
University of Marburg, Germany
jonathan@b-studios.de

HessPL, September 10th 2014

Why Encode Objects in an OO-Language?

Object Algebras [1]: Allowing modular defined *folds* by choosing a first class representation of algebras.

My *obj*.extend Library: Allowing modular defined *unfolds* by choosing a first class extensible representation of coalgebras.

[1] Oliveira, Bruno C. D. S., and William R. Cook. "Extensibility for the Masses." *ECOOP 2012–Object-Oriented Programming*. Springer Berlin Heidelberg, 2012.

Why do I need this, again?

Everytime you find yourself wishing that decorators would support **late binding** and could be arbitrarily **composed**:

$$\mathit{obj}.\mathbf{extend}$$

is what you are looking for.

An Example of Objects in Scala.

Two plain Scala traits:

```
trait Counter {
  private var i: Int
  def get: Int = i
  def inc: Unit = { i += 1 }
}
```

```
trait SkipCounter { self: Counter ⇒
  def skip: Unit = { this.inc; this.inc }
}
```

Static Mixin Composition in Scala ...

... allows:

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

... but it does not allow:

```
val c = new Counter { var i = 0 };
c extend SkipCounter
```

Second Class Traits!

... what can be achieved with *obj*.**extend**:

```
val c = unfold(Counter, 0);
c.extend(SkipCounter, ())
```

First Class Values!

Use Cases for *Dynamic Specialization* [2].

Incremental construction of objects performed by modularized builders

Adding methods for printing and tracing in order to facilitate debugging

Annotating objects with additional information, acquired after object creation

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

obj.extend Enables Dynamic Specialization by ...

... building on a **coalgebraic encoding of objects [4]:**

- **Interfaces** are encoded as an *interface endofunctor* `F`.
- **Implementations** are encoded as *coalgebras* `S ⇒ F[S]`.
- **Instantiation** is encoded by *unfolding* a coalgebra with an initial state to the greatest fixed point `Fix[F]`.
- **Objects** are encoded as *terminal coalgebras* over these functors.

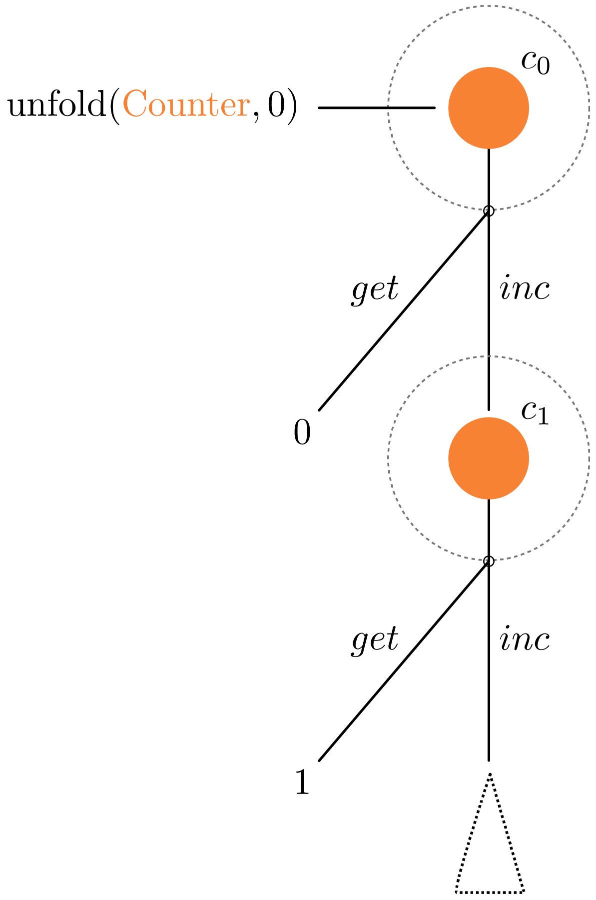
```
val Counter = (i: Int) ⇒ new CounterF[Int] {...}

val c = unfold(Counter, 0)
```

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

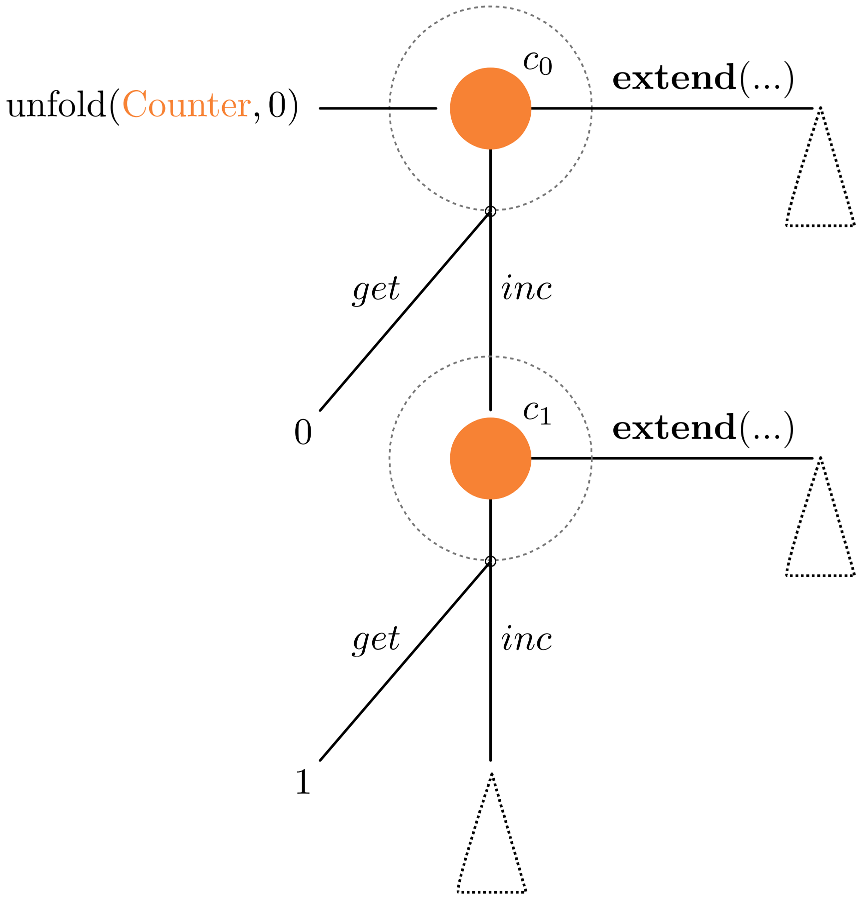
[4] B. Jacobs. *Objects and Classes, Coalgebraically*, pages 83–103. Springer-Verlag, 1995.

An Example of a Standard Terminal Coalgebra.




```
val c₀ = unfold(Counter, 0)
val c₁ = c₀.inc
```

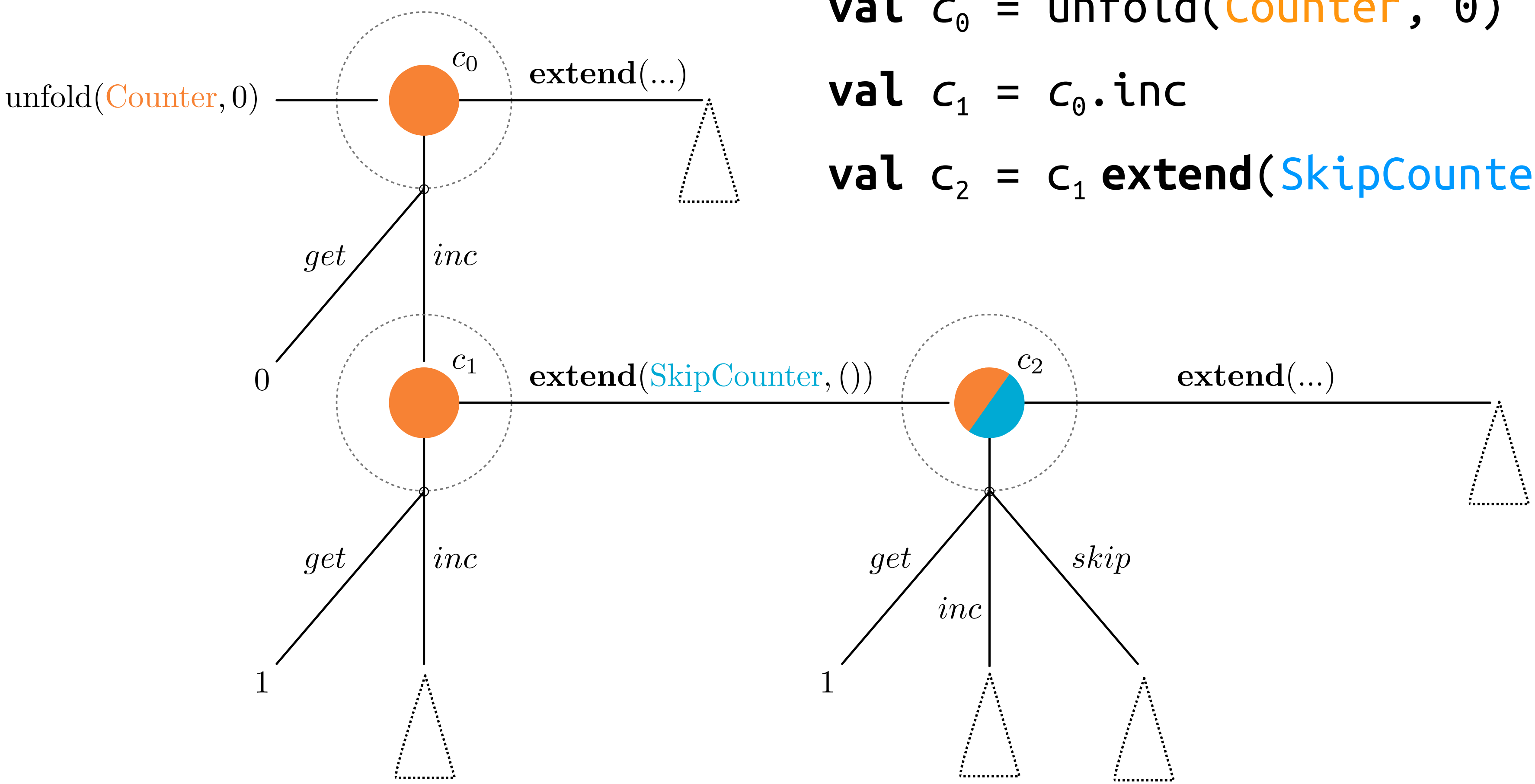
Same Example with *obj*.extend.




```
val c₀ = unfold(Counter, 0)
val c₁ = c₀.inc
```

Same Example with *obj*.extend.

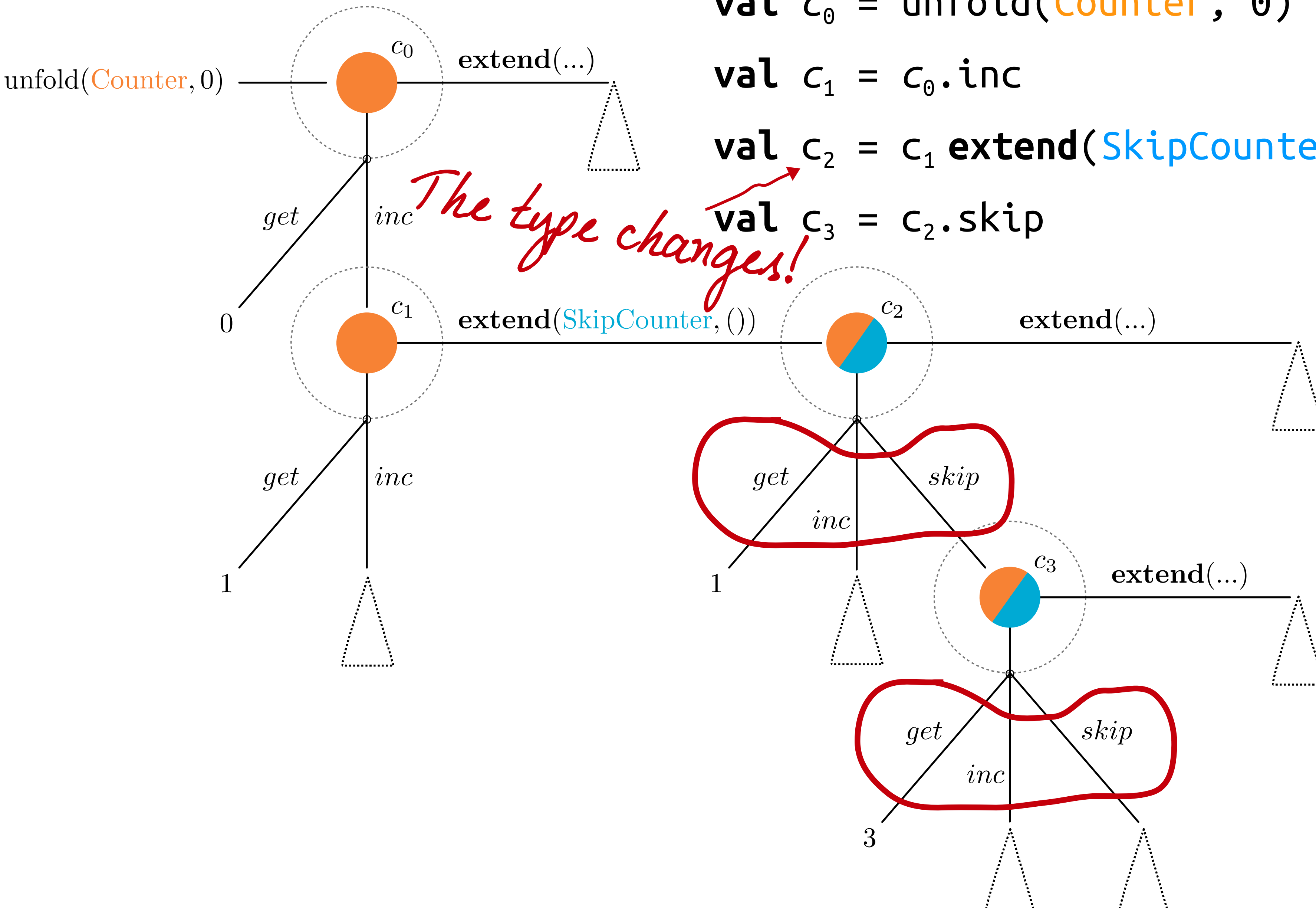
```
val c0 = unfold(Counter, 0)
val c1 = c0.inc
val c2 = c1 extend(SkipCounter, ())
```



Same Example with *obj*.extend.

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

The type changes!



obj.extend Enables Dynamic Specialization by ...

... defining the function `compose(co₁, co₂)` on coalgebras as

$$\texttt{compose}(\texttt{co}_1, \texttt{co}_2) \cong \texttt{s} \Rightarrow \texttt{mix}(\texttt{co}_1(\texttt{s}), \texttt{co}_2(\texttt{s}))$$

... implementing **`extend`** in terms of `compose`, thus

$$\texttt{unfold}(\texttt{co}_1, \texttt{s}_1)\ \textbf{extend}(\texttt{co}_2, \texttt{s}_2)$$

is implemented as

$$\texttt{unfold}(\texttt{compose}(\texttt{co}_1, \texttt{co}_2), (\texttt{s}_1, \texttt{s}_2))$$

And there is more to *obj*.extend...

- Allows *mutual dependencies* between coalgebras by encoding self-references.
- Allows accessing *private* slices of the state using lenses.
- Allows references to the extended base coalgebra, imitating *super-calls*.
- Allows *selective open-recursion* [3] by passing the current as well as the late bound self-reference.

⇒ Some of the extensions have been used to translate a subset of open jdk writers to the encoding.

[3] 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.

Conclusions.

We have shown:

– We can encode dynamic specialization of objects in Scala.

It seems:

– Object algebras can be usefully dualized.

Future work:

– Optimize performance to be practically useful.

– Develop a consistent and easy to use dsl.

– Investigate duality to object algebras formally.

Further Materials.

Slides:

http://files.b-studios.de/hesspl-slides.pdf

ICFP SRC Poster:

http://files.b-studios.de/icfp2014-poster.pdf

Mixin Composition:

https://github.com/b-studios/MixinComposition

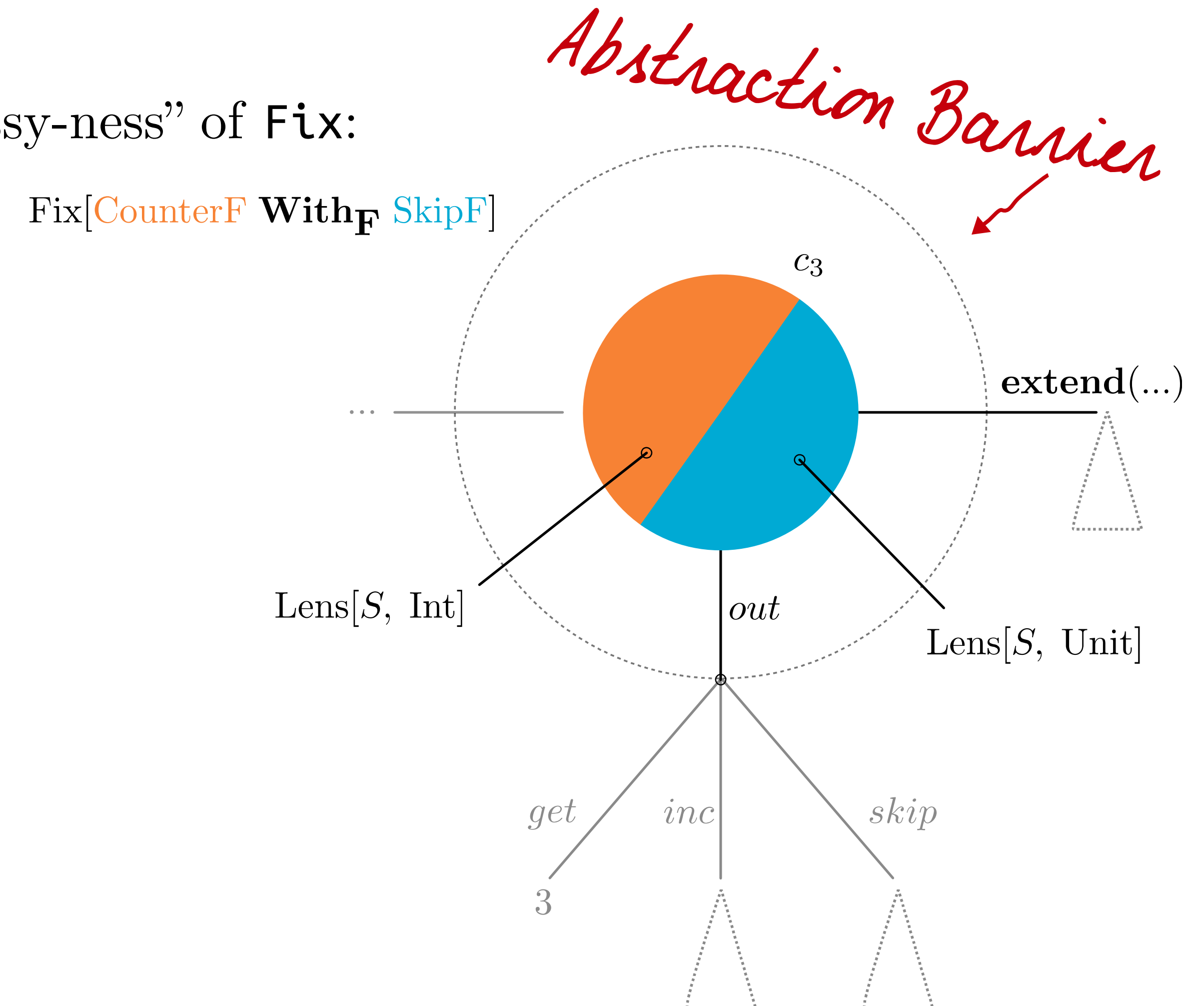
EOS

End of Slides, nothing to see here.

obj.extend Enables Dynamic Specialization by ...

... making use of the "first-classy-ness" of `Fix`:

- Novel method **`extend`** is added to `Fix`
- Original coalgebra and initial state are kept inside the closure of `Fix` but never revealed.




```
trait Fix[F[_]] {
  def out: F[Fix[F]]
  def extend[G[_], S2](co2: S2 ⇒ G[S2], state2: S2): Fix[F WithF G]
}
def unfold[F[_], S1](co1: S1 => F[S1], state1: S1): Fix[F] = new Fix[F] {…}
```