Revisiting the Cake Pattern: Scaling "Scalable Component Abstractions"

Paolo G. Giarrusso Jonathan Immanuel Brachthäuser
University of Tübingen, Germany

Abstract

The cake pattern was designed to support modular development combining mixins with ML modules, but it is criticized because mixins are not sufficiently isolated from each other's implementation. Indeed, as discussed by Gabriel, historically mixins were not designed to enforce isolation, but to support separating intrinsically orthogonal concepts in cooperative development scenarios.

We start investigating the issue and clarify which scenarios the cake pattern does succesfully apply to. We compare the cake pattern with an encoding of (recursive) hierarchical modules based on object composition instead of inheritance, that support fully separate modular development at the cost of more boilerplate.

We also suggest one could support separate modular development without additional boilerplate by combining the cake pattern with *private implementation inheritance*.

1. Introduction

Modularity helps to write and maintain complex codebases. One can use both *incremental modular development* (IMD) to develop and evolve new modules on top of existing ones, and *separate modular development* (SMD) to develop and evolve different modules in parallel once their interfaces have been established [5]. To fully enable separate modular development, a module system should support *separate compilation* [1], that is it should ensure that module linking succeeds whenever each module implementation typechecks against its interface (and implementations are available for each interface). Without this guarantee, sometimes it is necessary to coordinate evolution of different modules together. This coordination can be hard or impossible if different modules are maintained by independent developers, or simply if the codebase exceeds some complexity. Arguably, modularization then only allows a partial separation of concerns. However, this can still be useful for modules that are jointly developed and evolved, a somewhat restrictive but useful scenario we call *coordinated modular development* (CMD), for which mixins were designed [3].

Scala was designed to have good support for modular development by combining ML modules and mixins using what is now called the cake pattern [7, 8]; however, experience with the cake pattern revealed significant issues originally not anticipated. In fact, its biggest case study is the Scala compiler Scalac; but because of the experience, Scalac's rewrite Dotty is written minimizing the use of mixins.[1] Scala's type system is being formalized and its properties clarified, so we believe it is appropriate to also study in more detail how well Scala supports modular development, also using newer results on mixin modules [9].

In this short, paper we *sketch* the following contributions.

- We illustrate that the Scala cake pattern [7] does not support fully either SMD or IMD because adding members to a dependency can introduce errors at composition time.
- We point out that the cake pattern as-is can still be useful adequate for coordinated modular development, the scenario for which mixins were originally developed in object-oriented languages [3].
- We highlight that variants of the original pattern address some of these problems (at the cost of some boilerplate) and encode a subset of hierarchical ML modules *while still supporting recursive linking*.
- We point out that while mixin modules, in general, *can* support SMD, in an object-oriented context this would at least require extending Scala with a form of private inheritance, together with other typical constructs of module systems such as renamings and selective imports.

Throughout the paper, we ignore Java-style visibility control through *private* and *protected*, since it is less expressive

[1] `https://groups.google.com/d/msg/scala-language/WcnHXjAJaKg/i1XPUCpePrIJ`.

```
trait Trees {type Exp}
trait Eval { self: Trees with Normalize ⇒
  type Val
  val eval: Exp ⇒ Val
  val normalizingEval: Exp ⇒ Val =
    exp ⇒ eval(normalize(exp))
}
trait Normalize { self: Trees with Eval ⇒
  val reify: Val ⇒ Exp
  val normalize: Exp ⇒ Exp = exp ⇒ reify(eval(exp))
}

trait TreesImpl      extends Trees {...}
trait EvalImpl       extends Eval {...}
trait NormalizeImpl extends Normalize {...}

class Interpreter extends TreesImpl with EvalImpl
  with NormalizeImpl

object interpreter extends Interpreter
```

Figure 1. De- and recomposition into recursive modules using the cake pattern. Dependencies highlighted in *grey*.

than pure ML-style mechanisms and introduces further complications.

Some of these points are partially known to experts or touched upon in part outside the Scala literature [9], but not discussed in the academic literature on Scala, and we believe further research on them is needed.

2. Modularization by Inheritance: The Cake Pattern

To briefly illustrate the cake pattern, we use it to decompose a fictive normalizing interpreter into the three (recursive) modules Trees, Eval and Normalize (Figure 1). Following the cake pattern, dependencies to other modules are expressed using self-type annotations (such as *self*: Trees **with** Eval ⇒) and mixin composition is used to compose the slices and resolve the mutual dependencies.

The abstract type *Exp* is shared across all modules and the type *Val* is shared between Eval and Normalize. For instance, normalize only typechecks because the typechecker identifies Normalize.*Val* and Eval.*Val* thanks to the self-type annotations.

2.1 Limitations of the cake pattern

The cake pattern derives most of its modularity attributes from using mixin composition (as opposed to object composition), which is a form of (multiple) inheritance.

Using the cake pattern, composition of slices only requires a minimal amount of boilerplate. No explicit wiring of the type and value members of the involved components is necessary.

However, in Scala, module implementations are only hidden from each other in the modules themselves, but neither in their composition nor to their clients. For instance, any code in the body of *interpreter* in Figure 1 would unavoidably see the concrete definitions for types *Exp* and *Val* and any additional methods from the implementations, unlike in MixML [9] and Backpack [5], even if those concrete definitions are not intended to be part of the public API of the module. Worse, a module might typecheck or not depending on hidden implementation details of its dependencies, complicating IMD and SMD. Hence:

- Implementation details and dependencies to other modules are not hidden from clients; importing a module reexports it.
- Linking separately-developed well-typed modules can lead to linking errors, at least due to multiple possible variants of name conflicts. Such conflicts can be avoided for CMD or even in IMD, but prevent SMD.

Access modifiers can alleviate some of these problems, but only in a CMD scenario, since they must be added to individual modules and not at combination time.

Moreover, using self-type annotations the cake pattern forces the use of inheritance rather than object composition. This hinders having multiple instantiations of modules, which can be especially important if they have internal state.

2.2 Tradeoffs

When modules are designed together, are expected to coevolve and need to share implementation details the cake pattern can be appropriate. While using self-type annotations with interface traits to communicate dependencies allows one to (statically) recompose a system from different implementation slices, it entails tight coupling between the slices.

3. Modularization by Object Composition: Encoding Hierarchical Modules

As always in object-oriented programming, if loose coupling is desired, it might be advisable to use object composition instead of class composition (via inheritance) [6]. At the same time, it is folklore that abstract members (or other forms of parametrization such as constructor arguments) can be used to express dependencies to other modules [2].

In this tradition, Figure 2 shows an alternative to Figure 1 using an encoding of hierarchical modules instead of the cake pattern. A similar encoding was sketched by Odersky and Zenger [7], but without discussing the modularity tradeoffs. To express sharing and resolve recursive dependencies on the term level, laziness (or alternatively mutable state) is sufficient. However, as soon as abstract type members are shared across different modules, sharing constraints have to be introduced to allow the modules to collaborate. Since the type $t.Exp$ is shared between module Eval and module Normalize, in our example, sharing constraints of

```
trait Eval { outer ⇒
  val t: Trees
  val n: Normalize { val t: outer.t.type }
  type Val
  val eval: t.Exp ⇒ Val
  val normalizingEval: t.Exp ⇒ Val =
    exp ⇒ eval(n.normalize(exp))
}
trait Normalize { outer ⇒
  val t: Trees
  val e: Eval { val t: outer.t.type }
  val reify: e.Val ⇒ t.Exp
  val normalize: t.Exp ⇒ t.Exp =
    exp ⇒ reify(e.eval(exp))
}
class Interpreter { outer ⇒
  object t extends TreesImpl
  object e extends EvalImpl {
    val t = outer.t; val n = outer.n}
  object n extends NormalizeImpl {
    val t = outer.t; val e = outer.e}
}
object interpreter extends Interpreter
```

Figure 2. De- and recomposition into recursive modules using object composition and sharing constraints. Dependencies are highlighted in *grey* .

the shape {**val** t: $outer.t$.**type** } are necessary to assert that both dependencies use the very same instance of Trees.

Sharing of abstract type members can be performed on two different levels of granularity. First, we might introduce a type binding (**type** $T = self.T$) for every type that is shared between the modules. Second, as in the example, we might introduce a type refinement to a singleton type (**val** dep: $self$.**type**) for every module that declares types which are shared between the dependencies. The first is more verbose while the latter is more restrictive as it immediately implies the necessary wiring of the modules. This is reminiscent of the granularity of required interfaces in the cake pattern where one can either use self-type annotations to depend on a whole interface or abstract members for more fine-grained control.

3.1 Limitations of hierarchical modules

The encoding of hierarchical modules inherits most of its attributes from using object composition.

Wiring boilerplate The major advantage of mixin modules is automatic wiring of abstract and concrete type and value members. In contrast, to compose hierarchical modules we have to:

1. manually wire the dependent modules, that is, for every module provide references to its dependencies. For n modules, this amounts to $\Theta(n^2)$ assignments. The wiring is local to the composing module and has to be performed once per composition.
2. manually constrain the involved types of dependent modules, that is, for every module and every shared abstract type specify a sharing constraint. For n modules and m abstract types, this amounts to $\Theta(mn)$ sharing constraints. The sharing constraints are local to the corresponding module definition and do not require any additional boilerplate on composition. When using singleton types for sharing constraints this amounts $\Theta(n^2)$ type bindings.

When refining dependencies in subclasses of a module (e.g. the implementation of the module) sharing constraints have to be repeated. Just like self-type annotations have to be repeated when inheriting from cake slice.

Manual reexporting As always when using delegation over inheritance reexporting members has to be performed manually [4]. This is an old problem and, for instance, many generative solutions like macros or IDE support exist.

Privacy While protected state and methods are automatically shared in a flat cake, additional code is required in the nested cake solution to exhibit and share members that are not part of the public interface.

3.2 Tradeoffs

The encoding of hierarchical modules supports IMD and SMD at the cost of wiring verbosity and some restrictions in CMD. Yet, since the encoding is embedded in Scala as a host language, it interacts with other (object oriented) features like late-binding and overriding in specialized modules. This aspect goes beyond the expressivity of the ML module system or of MixML.

4. Conclusion, Discussion and Future Work

We discussed the complementary strengths of the cake pattern and the encoding of hierarchical modules and clarified when the cake pattern is still appropriate. In this section, we sketch possible hybrids and suggest future work.

4.1 Combining the two modularization strategies

Using self-type annotations forces different cake slices to be mixed together. But sometimes, even if we use an encoding of hierarchical modules, we can still use mixins at composition time to reduce the amount of code needed to perform the wiring, as in the following examples:

```
object hybridComposition₁ {
  object evalCake extends TreesImpl with EvalImpl {
    val t = this; val n = normalizer}
  object normalizer extends NormalizeImpl {
    val t, e = evalCake}
}
object hybridComposition_2 extends TreesImpl
  with EvalImpl
  with NormalizeImpl {val t, e, n = this}
```

Note how $hybridComposition_2$ uses mixin compositions to compose the different implementation slices. The advantage is that, unlike with self-type annotations, here the client side can choose which composition mechanism to use, and avoid using mixins whenever composition conflicts arise. We leave an investigation of this possibility for future work.

4.2 Historical roots

As discussed, the cake pattern has issues with separate modular development. But it might be unreasonable to expect otherwise also for important *historical* and *cultural* reasons, that relate to the underappreciated divide that Scala sits across. It is often said that Scala tries to bridge OOP and FP. But Gabriel [3] might imply that Scala tries to bridge across *incommensurable* communities with different goals, assumptions and technical language, such that even communication across them is extremely hard. Here we only offer informed speculation.

In particular, mixins (and more, in general, we believe many of the aspects that characterize OO languages, such as inheritance) were invented by a community working on object-oriented *programming systems* to support *cooperative* modular development. Gabriel [3] argues the later community working on object-oriented *programming languages* had incommensurable goals, in particular outlawing bad programs rather than just supporting good ones. In particular, the guarantees offered by *separate* modular development would have made little sense in the programming systems community. In this light Scala mixins (as opposed to MixML mixins) appear to us technically closer to the goals of the systems community, because they don't support SMD (even though, of course, Scala's type system is a giant step in the "programming language" direction).

Overall, we conjecture that Scala is not just trying to bridge across programming language research on object-oriented and functional languages, which after all are *commensurable* communities, but also across the *incommensurable* systems and language communities. Moreover, we conjecture this incommensurability explains some of the conflicts around Scala.

4.3 Future work

ML modules are subtle and Scala's variant is no exception. In this short paper we clarified some issues with Scala's modularity support; to conclude, we propose a few issues to focus future research.

Clarifying encodings of ML modules Odersky et al. [8] claim *νObj* can encode ML modules without details. While examples using ML modules typically translate to Scala, no complete encoding has been formalized and proved correct, and a few details are unclear. Combining features from ML modules with mixins is even trickier: supporting SMD with the cake pattern would require adding a mixin to a type after sealing it, but this appears impossible since only instantiated objects can be "sealed". Also, linking recursive modules using aggregation requires setting up a cyclic graph without using mutable variables (since those cannot be used in types); using **val** members can cause issues with initialization order, using **object** works but seems restrictive, and using **lazy val** appears currently robust but is not supported in Dotty as it appears unsound in general.[2] MixML [9] deploys linear types to express assign-once reference and address a very similar problem, a significant complication. It's unclear to us whether Scala can offer a fully satisfactory solution.

Extending Scala to support SMD Separate modular development and mixin modules have been combined in other systems like MixML [5, 9]. Could we achieve the same by extending Scala? Could we do that without affecting its core typesystem? Can we combine SMD with Scala's additional OO features missing from MixML, such as implementation inheritance with open self-recursion?

We conjecture that translating Backpack's thinning step would correspond to introducing some variation of C++ private inheritance. E.g., in Figure 1 *interpreter* should inherit only privately from TreesImpl but publicly from Trees, hiding any members of TreesImpl not in Trees.

More generally, one might need integrating other typical constructs of module systems, such as renamings and selective imports, and allowing to use them to control private inheritance.

Acknowledgments

We'd like to thank Tillmann Rendel, Klaus Ostermann and Scott Kilpatrick for helpful discussions.

References

[1] L. Cardelli. Program fragments, linking, and modularization. In *POPL*, pages 266–277. ACM, 1997.

[2] M. Fowler. Inversion of control containers and the dependency injection pattern, 2004. URL `http://www.martinfowler.com/articles/injection.html`.

[3] R. P. Gabriel. The structure of a programming language revolution. In *Onward!*, pages 195–214. ACM, 2012.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software (Adobe Reader)*. Pearson Education, 1994.

[5] S. Kilpatrick, D. Dreyer, S. Peyton Jones, and S. Marlow. Backpack: Retrofitting Haskell with interfaces. In *POPL*, pages 19–31. ACM, 2014.

[2] As discussed in `http://www.scala-lang.org/blog/2016/02/17/scaling-dot-soundness.html`.

[6] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA*, pages 214–223. ACM, 1986.

[7] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57. ACM, 2005.

[8] M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A nominal theory of objects with dependent types. In *ECOOP*, pages 201–224. Springer, 2003.

[9] A. Rossberg and D. Dreyer. Mixin' up the ML module system. *ACM TOPLAS*, 35(1):2:1–2:84, 2013.