Haskell Cake

How to bake reusable components

Oleg Kiselyov
University of Tsukuba, Japan
oleg@okmij.org

Abstract

Scala shines in the construction of *large-scale* programs from separately developed and tested, extensible components. Scalable component programming comes from a particular combination of Scala features: abstract type members, explicit self-types and modular mixin composition. This combination has come to be known as Scala Cake pattern. The pattern particularly proves its worth during the evolution of large code bases. It becomes compelling in complex cases of aggregating mutually dependent but separately compiled extensible components.

What is this cake pattern, freed from the OO terminology? Why haven't we come across it before? Can we reproduce its success in building extensible components in Haskell? Does it alleviate the unabating library versioning problems? How convenient is scalable component programming in Haskell and how to make it syntactically sweeter?

The paper answers these questions on two case studies demonstrating how to reproduce the winning combination of Scala features in Haskell. Although Haskell modules are not extensible components, scalable component programming is nevertheless possible. We show how to use type classes as interfaces (component signatures) and how to systematically replace hard links – references to components and their types and values by name – with references by interfaces. The Haskell Cake pattern is compelling in sufficiently complex cases, especially mutually dependent extensible components; otherwise, ordinary closures suffice.

1. Introduction

Modularity is one of the key principles of software engineering [7]. Ideally a software system is put together from separately developed and tested, and easily interchangeable parts with clear interface. We call such parts, which provide a set of related data types and operations on them, components. They are also called packages, structures or modules – however, modules in Haskell have a specific meaning as a unit of compilation. A typical component is the module Data.Dynamic, providing a data type Dynamic and several operations on it such as toDyn and fromDynamic. The exact representation, the data constructors, of Dynamic is not exposed and so Dynamic values can only be created and transformed through the functions provided by the component. Dynamic relies on data types and operations of another component, Data.Typeable. Components like HTTP provide many datatypes, require many other components for their functionality, and are comprised of many modules.

Most of Haskell components however are not extensible and composable. For example, to use Data.Dynamic with a different version of Typeable we at the very least must recompile the former (which may set off a cascade of further recompilations). The reason is import Typeable in Dynamic, referring to the module Typeable by its name. There is no other way: Typeable is described by what a particular module Data.Typeable implements rather than by what that module must conform to. In contrast, Scala Cake lets dependencies between components be specified in terms of their interfaces. Therefore, a separately compiled component, as it is, may be later linked with any implementation of the required interfaces. Extending a Scala component with more types or operations will not break its dependents. Not only are the sources of the dependent components unaffected by the extension, but also their compiled binaries. Using two different versions of a base component within the same program is straightforward in Scala and nearly impossible in Haskell.

The success of Scala in large-scale component development comes about because, argued Odersky and Zenger [6], the language has features to satisfy the necessary requirements for scalable component programming:

- separate interface and implementation: The interface enumerates the (abstract) types and operations supported by the component. Several components may implement the same interface, with their own internal operations and realization of the abstract types.
- abstraction over required functionality: If a component needs the functionality of other components, the latter should be referred to by their interfaces rather than by names.
- coarse-grained, largely implicit linking: The user should be able to build programs by specifying components or large component assemblies, wasting no time on matching each and every required and provided operation.

The first requirement is the well-known encapsulation, ensuring representation independence [8]: the ability to change the representation of an abstract data type without affecting the rest of the program. The second requirement is less known. Odersky and Zenger are to be credited for motivating, drawing attention to and clearly stating the need to abstract over the required functionality.

Odersky and co-workers [5, 6] explain in detail which features of Scala work to satisfy which requirement: nesting of classes for aggregation and encapsulation; abstract type members with upper bounds for abstracting provided data types; self-annotations for abstracting the required functionality; mixin composition for matching provided and required functionality. This particular combination of features came to be called the *Scala Cake* pattern. Many explanations of the cake pattern have been written in various blogs[1] – alas, Scala-specific, often using too simple examples, making the pattern contrived and over-broad, and invariably making it appear as a Scala trick. In the result, the pattern is hard to understand, especially outside Scala and object-oriented programming. How it can be reproduced in Haskell is a good question.

Furthermore, it is not clear if Haskell can fulfill the requirements for the scalable component programming at all. Haskell does not separate module interface and implementation. Unlike ML, Haskell has no notion of module signatures, or module types. Haskell supports ascribing a signature to a single definition but not to a module (a collection of definitions and type declarations). Also unlike ML, Haskell does not support parameterized modules (functors), and so seemingly unable to abstract over the required functionality. To refer to operations in other Haskell modules, we have to `import` them by the name of the implementation file rather than by the interface. Odersky [5, 6] calls such references "hard links". Hard-wiring of the module names is responsible for *numerous* versioning problems, an unremitting bane of Haskell installations[2]. What self-types and mixins of Scala, the ingredients of the winning pattern, may correspond to in Haskell is a puzzle. It may seem that for large-scale development, for assembling programs from components, Haskell is severely lacking.

Contributions We demystify the Scala Cake pattern and explain it in terms of functional languages, showing its representation in Haskell. We demonstrate the systematic way of eliminating hard links among (mutually) dependent components, using type classes for module interfaces and higher-rank types for linking by interface rather by name, abstracting over implementation details and private state. Mutually dependent components are structured into non-recursive, separately compiled and extensible modules related only by the common interface. We call this pattern Haskell Cake.

Like Odersky and Zenger [6], we use two case studies to demonstrate scalable component programming. An alternative exposition would prove that core Scala features that underlie the Cake pattern are implementable in Haskell, by developing a formal translation from the Scala core calculus νObj to a Haskell core calculus, and proving its properties. We do not take this alternative since our goal is not to encode OOP in Haskell but to demonstrate developing scalable components in *idiomatic* Haskell.

Along the way we show why we have not come across the cake pattern in Haskell before. This is because for simple extensible components, ordinary closures suffice. It is only in complex case like those in §5 or §6 that the cake pattern becomes compelling. Therefore, we have to use a moderately complex running example, making it even more complex at the end. Clarifying when closures work to implement components, when they stop working and how to retrofit them into the cake pattern, §4.6, is another contribution.

The end result, like in Scala, is scalable component programming. The main ingredients of the Haskell cake pattern are:

- type classes with associated types as component's interfaces
- higher-rank types (existentials or universals) to refer to a type by its interface it satisfies rather than by its name
- Writing components as traits (mixins). The defining property of a trait is being "upwards closed": if an interface is specified by a type class `C` and a component `T` implements `C` (that is, `T` is an instance of `C`) then any aggregate `T'` that incorporates `T` also implements `C`. We call `T'` as incorporating `T` if it is possible to project out and update `T` from `T'`.

As any principles, when stated they appear trivial; the difficulty is in applying them. The paper describes how to systematically apply these principles, how to gradually eliminate hard links and develop traits.

Scala Cake is a pattern: a way to write code. It is up to the programmer to follow the pattern; Scala per se does not automatically lead to reusable and extensible components. Likewise, our set of patterns is a style of writing Haskell code, to make libraries (components) extensible and resilient to version changes.

The structure of the paper is as follows. §2 introduces the running example and shows how the 'traditional' implementation (the implementation style seen in the standard and other libraries, e.g., `Data.Data`) leads to the reusability failure and versioning problems. A simple addition of a debug printing to one operation sets off the massive cascading code duplication. We fix the problems by gradually introducing the ingredients of the Cake pattern in §3 and §4, replacing hard links with soft ones. §3 explains writing a separate interface specification and referring to a component by its interface using universals. §4 finishes eliminating hard links, using existentials, and demonstrates component aggregation relating it with traits. In §4.6 we investigate to which extent closures suffice for component programming. §5 shows off the full power of traits for building components by aggregation. The second case study of mutually dependent and yet separately extensible components providing operations and types for each other is described in §6. This example, extracted from the Scala compiler, was at the center of [5, 6]. It turns out the Haskell Cake pattern lets us implement the example in idiomatic Haskell, with no OO features.

The complete code accompanying the paper is available at `http://okmij.org/ftp/Haskell/ScalaCake/`. Therefore, when presenting code the paper will show only salient parts, referring to the full code for details. The online code also includes a number of files demonstrating features of Scala such as abstract classes, traits, mixin composition, and their emulation in Haskell.

2. Hard links and the composability failure

This section introduces the running example and its idiomatic implementation as two Haskell modules, depending on each other. These modules fall short of reusable components. The module dependence is expressed through hard links, by referencing the names of the exported values and types. Therefore, when one module is extended with debug printing while preserving the interface, the code of the dependent module has to be duplicated. We will observe the reusability failure and the all-to-common in Haskell versioning problem. The follow-up sections will re-implement the example to fix these problems.

2.1 Publish-subscribe example

Our running example is a publish-subscribe system comprised of two components, subjects and observers. A subject accepts subscriptions from observers whereas an observer accepts notifications from subjects. The operation `publish` on a subject will notify all subscribed observers; the operation `withdraw` will cause an observer to unsubscribe itself from all subjects it has subscribed to previously. Subjects and observers are defined, quite symmetrically, in terms of each other. After we implement the library for publishing and subscribing, we use it to define sensors as concrete subjects and displays as observers.

[1] `http://debasishg.blogspot.in/2013/02/modular-abstractions-in-scala-with.html` `http://jonasboner.com/2008/10/06/real-world-scala-dependency-injection-di/` `http://blog.rintcius.nl/post/di-on-steroids-with-scala-value-injection-on-traits.html`

[2] `http://www.well-typed.com/blog/9` `http://cdsmith.wordpress.com/2011/01/16/haskells-own-dll-hell/` `http://www.google-melange.com/gsoc/proposal/review/google/gsoc2012/phischu/1` More articles can be found by doing Haskell-Cafe or Google search for "Haskell dependency hell"

The example is an extended version of the Scala cake case study in [6, Sec 3], illustrating what some call family polymorphism (a family of components, two in this case, which vary covariantly). The case study in that paper is too simple to make the point in Haskell: an observer with a single method notify can be represented as a simple closure. We keep to the design decisions of the Scala example, and implement them in idiomatic Haskell. Publish-subscribe idiom is quite common: e.g., it occurs in GHC run-time system, GHC.Event. Our implementation in this section has many similarities with the Event module.

We start with the module SubObs1 that exports the types Subj and Obs of subjects and observers; the constructor functions newSubj and newObs; the operations subscribe, publish and withdraw. We describe the salient points of the module below; for the complete code, see the accompanying file `SubObs1.hs`. The reader may feel unease with our description of the module interface: partly in English, spread out through the whole section and intermingled with implementation. This is typical but not inherent in Haskell; §3 tells how to write a component interface in Haskell itself and have the compiler check the implementation's compliance.

EXERCISE 1. *Subjects and Observers could be implemented in separate modules, which have to be mutually dependent. What additional operations should be exported then?*

Before we can flesh out the interface of SubObs1 we have a decision to make. The principle of representation independence dictates that we hide the implementation of the subscription list and export only the type Subj with publish and subscribe operations, but not its realization. Concrete subjects, like the sensor below, have richer structure (e.g., the state of the sensor) with more operations. We have to decide how the bare Subj gets incorporated or extended into concrete subjects. There are two ways: a concrete subject either incorporates Subj, or is an instance of Subj. We will try both, exploring their trade-offs. In this section, we choose Subj as a family (a polymorphic data type) abstracting over concrete subjects and their concrete data. (In other words, Subj is an extensible data type, [2].) Subj operations such as subscribe will be polymorphic, uniformly applying to Subj a for any a. Such a design is conventional in Haskell, see Data.Tree for example. (We will come to regret this design of Subj; for example, it cannot express proxies, §5. §4 explores the alternative, less common design, part of the cake pattern.)

We choose the straightforward representation for Subj a, the type of a subject that accepts and maintains subscriptions from observers Obs a. (GHC.Event.EventManager has a similar implementation.)

```
data Subj a = Subj a (IORef [Obs a])

subjData :: Subj a → a
subjData (Subj x _) = x
```

Since the realization of Subj is not exposed, we have to export subjData to extract the data of concrete subjects.

An observer of Subj a, when notified, receives the subject, from which it can extract the concrete data. Obs therefore likewise define a family of types, matching Subj. An observer, too, maintains the subscription list, of subjects, to be able to unsubscribe from them. An observer also has a name, to be used as an identifier when subscribing or unsubscribing.

```
data Obs a = Obs{
  obsName :: String,
  notify     :: Subj a → IO (),
  obsSubjects :: IORef [Subj a]}
```

A state, private data of an observer, if any, is incorporated into the closure notify. We stress that SubObs1 exports only the types Subj and Obs but not their representation – giving us the freedom to change the representation, e.g., to collect subscriptions in a data structure other than list.

The internal operation subjSubscribe adds an observer to the list of subscribed observers

```
subjSubscribe :: Subj a → Obs a → IO ()
subjSubscribe (Subj _ observers) obs = modifyIORef observers (obs:)
```

and a similar subjUnSubscribe removes. The exported subscribe links a subject and an observer together

```
subscribe :: Subj a → Obs a → IO ()
subscribe subj obs = do
 subjSubscribe subj obs
 modifyIORef (obsSubjects obs) (subj:)

publish :: Subj a → IO ()
publish subj@(Subj _ observers) =
  readIORef observers ≫= mapM_ (\obs → notify obs subj)
```

and publish notifies all subscribed observers; withdraw is similar to publish, only applied to an observer. Since the representation of Subj and Obs is not exposed, we have to export functions to construct them:

```
newSubj :: a → IO (Subj a)
newObs :: String → (Subj a → IO ()) → IO (Obs a)
```

An example of using the SubObs1 library is the module SensorReader1, which implements a Sensor as a concrete subject and Display as a concrete observer.

```
module SensorReader1 (
  Sensor, Display,
  newSensor, newDisplay, changeValue,
  subscribe, withdraw) where        -- re-export from SubObs1

import SubObs1

type Sensor  = Subj Sensed
type Display = Obs Sensed
```

where the sensor state is

```
type Label = String
data Sensed = Sensed{label  :: Label,
                     senVal :: IORef Double}
```

The module SensorReader1 has to import SubObs1 since it refers, by their name, to the types Subj and Obs and functions defined therein.

Besides the operations common to all Subj, Sensor also lets us change its value, notifying all subscribed observers:

```
changeValue :: Sensor → Double → IO ()
changeValue subj nv = do
  Sensed{senVal= vr} ←subjData subj
  writeIORef vr nv
  publish subj
```

Once notified, Display prints the current value of the sensor:

```
newDisplay :: String → IO Display
newDisplay l = newObs l notify
 where notify subj = do
          let sdata = subjData subj
          v ← readIORef (senVal sdata)
          putStrLn $ unwords ["display", l, "sensor", label sdata,
                              "has_value", show v]
```

The main module SRTest1 imports only SensorReader1 and creates a few sample sensors and displays, subscribing them and changing the sensors' values.

2.2 Extending the publish-subscribe example

To demonstrate version problems, let us extend SubObs1 with debug printing: the publish operation will print the name of each observer before notifying it. The original version is not to be discarded; we even want to use it in the same project alongside the debugging version. The new version, called SubObs1d will be a nearly duplicate of SubObs1. The duplication is inevitable since SubObs1 hides the subscription list. We pay for representation independence with duplication. More annoyingly, by duplicating SubObs1 we duplicated the implementation of Obs, which had nothing to do with the the change in publish.

EXERCISE 2. *Had we implemented subject and observer in separate, mutually dependent modules as in Ex. 1 we would still need to duplicate the observer code when we extend the subject code. Why?*

EXERCISE 3. *To print the name of a notified observer, we could have added the debug printing to* notify *instead. Does this alternative design avoid the duplication problems?*

Recall that the name SubObs1 was hard-wired, in the import statement, in the implementation of the sensor library, SensorReader1. If the sensors are to use the debugging version of subjects, we have to change the source code of SensorReader1. If we wish to keep the original library, we are forced to make a copy. The debugging version SensorReader1d is identical to SensorReader1, differing only in the name of the imported SubObs1d.

On the bright side, the old SensorReader1 and the debugging SensorReader1d may be used side-by-side in the same application as Test1d demonstrates. We import both versions, with qualifications:

```
import SensorReader1  as SR
import SensorReader1d as SRd
```

Moreover, the type checker enforces the isolation. Sensors and observers created by different versions of the library cannot be confused: we cannot use SR.subscribe to subscribe sensors or displays created by SRd. On the downside, we cannot use displays created by SR to observe sensors of SRd – even though the sensor interface has not changed and the implementation of observers is identical between the two versions of the library.

We saw how a simple change, adding debug printing, had a ripple effect of code duplication. Because of hard links, making a new version of a module forces changes in all transitively-dependent modules. A large part of code base has to be duplicated.

3. Separating interface and implementation

Although the common way of writing Haskell libraries fails to produce flexible, reusable components, Haskell nevertheless has facilities for scalable component programming. We describe them by gradually fixing the problems with the running example we have just seen. This section explains how to write a separate interface specification for a Haskell component and refer to a component by its interface using universals. §4 finishes eliminating hard links between our subjects and observers, using existentials. It demonstrate component aggregation and relates them with traits.

We have seen that the reusability failure arises from referring to types and values of another library directly by their name rather than indirectly by their properties identified by interfaces. At first blush referring to another library by its type, or interface, seems impossible since Haskell modules, unlike, say, ML modules, do not have types. Seemingly we cannot write a specification for a library and then ask the compiler if a purported implementation conforms to it. The export list of a module is a poor abstraction: it has no types, no constraints, and is inseparable from the module. And yet, Haskell does let us write a library interface and check the implementation's conformance to it, and does let us refer to a library by its interface. Admittedly, the library interface in Haskell is more roundabout compared to signatures of ML.

Haskell type classes, with associated types, provide all we need from an interface, or a module signature: the ability to bundle type and value signatures, hence to abstract over a collection of type and value definitions. As an example, here is the interface of the publish-subscribe library SubObs1 from §2.1. Recall that the library defines subjects that accept subscriptions from observers, and observers that can be subscribed to, withdrawn from, and notified by subjects. The following pair of type classes abstract from concrete types of subjects and observers and state the operations in terms of so abstracted types.

```
class  SUBJ subj where
  type Observer subj :: * → *
  subjData  :: subj a → a
  newSubj   :: a → IO (subj a)
  publish   :: subj a → IO ()
  subscribe :: subj a → Observer subj a → IO ()

class  OBS obs where
  type Subject obs :: * → *
  obsName  :: obs a → String
  newObs   :: String → (Subject obs a → IO ()) → IO (obs a)
  withdraw :: obs a → IO ()
  notify   :: obs a → Subject obs a → IO ()
```

The method signatures are quite like the signatures of the functions exported from SubObs1, only instead of the concrete types Subj and Obs with the hidden implementation, the methods use type-class–constrained type variables subj and obs. In other words, SUBJ and OBS abstract over all possible implementations of the publish-subscribe functionality and define a family of implementations. The earlier Subj and Obj were actually type constructors, describing a family of subjects and observers parameterized by the state of a concrete subject. The type variables subj and obs are likewise type constructor variables. Thus SUBJ and OBS represent double-abstraction, over the implementation of the publish-subscribe mechanism and over the concrete state of subjects.

EXERCISE 4. *The associated types* Observer subj *and* Subject obs *establish one-to-one correspondence between the types of subjects and their observers. Such a design reflects the Scala example from [6, Sec 3] and* SubObs1*. Is such one-to-one correspondence necessary? Why couldn't we define the method* subscribe *with the signature* $OBS\ obs \Rightarrow subj\ a \to obs\ a \to IO\ ()$*? Or could we?*

The type classes SUBJ and OBJ are placed in their own module SubObsCl, which will play the role of the publish-subscribe interface. One implementation for the library is in the module SubObs2, which defines the same data types and operations as SubObs1. The module SubObs2 now constructively proves that these operations conform to the 'declared' publish-subscribe interface, by importing SubObsCl and defining the instances for SUBJ and OBS. For example:

```
instance  SUBJ Subj where
  type Observer Subj = Obs
  subjData (Subj x _) = x
  newSubj x = Subj x `fmap` newIORef []
  publish  subj@(Subj _ observers ) =
        readIORef observers  >>= mapM_ (\obs → notify obs subj)
  subscribe  subj  obs = ...
```

and the similar instance OBS Obs. The module SubObs2 exports just the types Subj and Obs – and overtly nothing else. The instances are exported implicitly.

EXERCISE 5. *Can we separate the implementation of SUBJ and OBJ into non-mutually dependent modules, perhaps after small adjustments to the SUBJ and OBJ interface?*

The new way of writing the publish-subscribe library, with the explicit interface specification, has hardly affected the dependent sensor-display library. In fact, the only change between the new SensorReader2 and the SensorReader1 from §2.1 concerns the import-export list and type signatures. Whereas SensorReader1 imported the implementation SubObs1 (by its name), SensorReader2 imports only the interface, SubObsCl. A Sensor is now a family of types, parameterized by the concrete implementation of subject (Display is similar):

```
type Sensor subj  = subj Sensed
type Display  obs = obs Sensed
```

The operations on Sensor and Display are polymorphic, with type class constraints spelling out the required interfaces:

```
newSensor   :: SUBJ subj ⇒ Label → IO (subj  Sensed)
changeValue :: SUBJ subj ⇒ subj Sensed → Double → IO ()
newDisplay  :: (SUBJ (Subject obs), OBS obs) ⇒
                  String  → IO (Display  obs)
```

For example, newDisplay will construct a Display given any implementation obs of the OBS interface provided that the corresponding subj is an implementation of the SUBJ interface. Again, the code for the above three functions is the same as the one in SensorReader1.

We stress that the required functionality, values and types, is now referred to indirectly, by the interface. We have called a hard link a direct reference to a type or a non-overloaded function by name. We may call a reference to a function or a type through interface, or type-class constraint CTX, 'a soft link'. In other words, a soft link, rather than referring to a particular type name T, contains $\forall t.\ \ CTX\ t \Rightarrow t$.

The concrete implementation of the publish-subscribe interface, SubObs2, is imported only in the main module, SRTest2, which does the final assembly of components.

```
main = do
  s1 ← newSensor "sensor1"  :: IO (Sensor Subj)
  d1 ← newDisplay "d1"
  subscribe  s1  d1
  ...
```

Our Sensor subj and Display obs are polymorphic over the concrete implementation. Eventually we have to specify which implementation we want – by giving the type annotations as in the code above. Luckily Haskell saves us from writing such annotations all over the place: for example, we did not have to annotate the construction of a Display obs in the above code because GHC inferred the concrete type of the obs. One may wish the type of subj were inferred too, since there is only one suitable type in scope that satisfies the needed constraint of being an instance of SUBJ – namely Subj. (The language $\mathcal{G}$ of concept-generic programming [10] could do such an inference.) One could further wish GHC determine which types in scope fit the constraints; and if several found, ask the user to choose one.

The pay-off comes when building a new implementation of the publish-subscribe library, with debug printing. As before, §2.2 we copy SubObs2 to SubObs2d and change the implementation of publish to print out the name of each observer as it is notified. Unlike §2.2 we no longer copy the dependent code, SensorReader2 in our case, since it is polymorphic over the publish-subscribe implementation, and, hence, imports no concrete implementation module.

EXERCISE 6. *If the answer to Ex. 5 is yes, could we finally avoid duplicating the observer implementation when we add debug printing to the subject implementation?*

The main module, SRTest2d may mix-and-match the plain SubObs2 and debug SubObs2d versions by importing both of them (with the different qualifications, SO vs SOd) and using the type annotations to indicate which version is desired:

```
test  = do
  s   ← newSensor "sensor1"   :: IO (Sensor SO.Subj)
  sd  ← newSensor "sensor1d"  :: IO (Sensor SOd.Subj)
  d   ← newDisplay "d1"
  dd  ← newDisplay "d2"

  subscribe  s  d         −− non−debug version of sensor and display
  subscribe  sd  dd       −− debug version of sensor  and display
  −− subscribe s dd  −− version confusion:  type  error
```

Some annotations (on Displays) can be omitted since they are inferred. As before, the type checker enforces the isolation, preventing an observer from one version of the library to subscribe to a differently-versioned subject.

Thus separating interface and implementation and referring to required libraries by their interface (soft links) predictably eliminated the version problems. Although the common way of writing Haskell libraries may indicate otherwise, Haskell is not deficient when it comes to scalable component programming. We shall see more evidence below as we continue improving our design.

4. Traits

We have demonstrated one way of writing composable components. The sensor library SensorReader2 can be composed with any implementation of the publish-subscribe interface SubObsCl. There is still much room for improvement. Although SensorReader2 is no longer tightly coupled with the implementation of subjects and observers, the latter remain coupled among themselves. Therefore, building the debug version of Subj forced a new version of Obs, with the resulting code duplication. Further, a single observer cannot subscribe to two differently-typed subjects but with the same type of data to observe. In this section, we solve these remaining composability challenges and illustrate the cake pattern in full. Incidentally, the Scala case study of publish-subscribe [6, Sec 3] has the same problems, which are overcome in the second case study of that paper.

4.1 Untangling the observer interface

The tight coupling of subject and observer implementations has already become clear if you did Ex. 5. The operation subscribe that links a subject and an observer together has to have access to the internal representations of both. If we make subjects and observers separate components (which hide their internal representation), we have to add an operation subscribed to the OBS interface and likewise modify Subj. OBS will now read:

```
class  OBS obs where
  type Subject  obs :: ∗ → ∗
  obsName    :: obs a → String
  newObs     :: String  → (Subject obs a → IO ())  → IO (obs a)
  subscribed :: obs a → Subject  obs a → IO ()
  withdraw   :: obs a → IO ()
  notify     :: obs a → Subject  obs a → IO ()
```

The method subscribed implements observer's part of linking, remembering the reference to the subject being subscribed to.

EXERCISE 7. *How come we did not have such a method before? Hint: check the implementation of SubObs2.subscribe.*

The addition of subscribed to OBS and the corresponding changes to Subj let us separate their implementations into non-recursive modules – as you must have done when solving Ex. 5. The remaining problem is the tight coupling in their *interfaces*, which is apparent from the signature of subscribed and notify. Both OBS methods receive a subject as the second argument, whose concrete type is uniquely determined from the concrete type of obs. A particular observer hence refers to the concrete subject type. The hard links prevent an observer from subscribing to two subjects that publish the same data but have different types of their state or different implementations, or versions.

EXERCISE 8. *How come adding debugging printing to a subject changes its type? Hint: a new version of a Haskell module has types incompatible from the types of the old version.*

The signature of notify betrays another flaw: notify receives the complete subject and hence may observe its entire concrete state and arbitrarily modify its mutable parts. The interface offers no way to restrict access to subject's state.

To avoid the hard link, we introduce the existential (to be refined in §4.2, that's why the name is primed)

```
data ASubj' a = ∀subj. SUBJ subj ⇒ ASubj' (subj a)
```

which lets us re-write the subscribed and notify signatures as

```
subscribed :: obs a → ASubj' a → IO ()
notify     :: obs a → ASubj' a → IO ()
```

EXERCISE 9. *Can we eliminate the hard link to a subject type without existentials, thus avoiding annoying packing/unpacking operations?*

As the name suggests, ASubj' dat is *a* subject with the concrete state a. The method subscribed refers to the subject by the interface, by the SUBJ subj constraint. Since the concrete type of the subject is hidden within the existential, the same observer may subscribe to subjects of different representations. We call ASubj' a a "generic subject".

A few easy improvements spring to mind. First, notify is invoked to let the observer know of the changed state of the subject. The method receives the complete subject, from which it should extract the subject's state and then published data (and avoid looking let alone modifying other parts of the subject). Since notify should only apply the subjData operation on the received subject, this application may just as well be carried out by the caller of notify; the existential is hence eliminated, replaced with the result of subjData subj:

```
notify     :: obs a → a → IO ()
```

The caller of notify can also extract the part of subject's state relevant for the observer. In other words, notify will receive not the entire state of the subject but only a (small) view of this state – the published data. We no longer have to worry of notify seeing too much of subject state or modifying it. Finally, the polymorphic a in notify's signature indicates that an observer processes the published data uniformly, regardless of their type – which is a restriction. For example, an observer that logs the notified data has to require a to be in the Show class. For the sake of flexibility, we no longer parameterize the observer by the type of published data; we merely state that the type of the observer determines the type of data (cf. the design of Haskell collections, which was one of the motivations for associated types [1]). We thus arrive at the following new OBS interface (see file `SubObsTr.hs`):

```
class OBS obs where
  type ObsData obs :: *
  obsName    :: obs → String
  subscribed :: obs → ASubj (ObsData obs) → IO ()
  withdraw   :: obs → IO ()
  notify     :: obs → ObsData obs → IO ()
```

The type of observers is no longer parameterized by the concrete state of subjects. Rather, associated with each obs type is the type ObsData obs of observed data – the data that a subject lets observe. There is no newObs method any more (the reason becomes clear only in §5, but the reader is encouraged to guess).

We have eliminated hard links from an observer type to the subject type. The new observer refers to the subscribed subject by the SUBJ interface and by the type of data it publishes, that is, by a public *view* of the subject state.

4.2 Untangling the subject interface

Having softened the links from observers to subjects, we do the same for the links from subjects to observers. As the changes mirror those in §4.1 we show only the final result, the new SUBJ class, which is quite symmetric to the new OBS and is also part of the SubObsTr module:

```
class SUBJ subj where
  type SubjData subj :: *
  publish      :: subj → SubjData subj → IO ()
  subscribe '  :: subj → AnObs (SubjData subj) → IO ()
  unsubscribe' :: subj → AnObs (SubjData subj) → IO ()
```

The old subscribe is replaced with subscribe' and unsubscribe'. The method subscribe' differs in semantics from subscribe: whereas the latter links a subject and its observer *together*, subscribe' only affects the subject, adding a new observer to its subscription list. The operation unsubscribe' does the reverse.

EXERCISE 10. *How come we did not have these methods before? Hint: we didn't? Hint: answer Ex. 7.*

The methods subscribe' and unsubscribe' take an observer as an argument – a *generic* observer, which refers to an observer by its interface rather than the concrete type:

```
data AnObs dat =
    ∀ obs. (OBS obs, ObsData obs ~ dat) ⇒ AnObs obs
```

Any observer that accepts dat can subscribe to a subj that publishes dat.

Recall that in §2.1 we defined the subject type to be a family of types subj a polymorphic over the concrete state a of the subject. Each member of the family has the same realization (of the subscription list and operations on it) but a different state type. The new observers in §4.1 are no longer interested in the whole concrete subject state: an implementation of OBS refers to a subject only by the type of the published data. Therefore, the new subj exposes only that type, as SubjData subj. The full concrete state is an implementation detail and should be hidden.

This seemingly small change of perspective – the subject interface exposing only the type of published data – is momentous. First, the subjData method from SUBJ of §3 that returned the concrete state is gone.

EXERCISE 11. *Why was newSubj a method of SUBJ before, but not now?*

Since a subj is no longer parameterized by the concrete state it does not have to incorporate any concrete state, reversing the implementation decision made in §2.1. The new subj could stand only for the implementation of the subscription list and operations on it. Concrete subjects will incorporate subj, rather than being incorporated into it. 'Subject' became a trait, as we shall see next. This alternative design is more expressive, permitting subjects with several subscription lists. It also lets us implement proxies, subscription aggregators (see §5). The main attraction of the alternative design for us here is that it lets us distinguish subjects by the data they publish rather than by the data they possess.

Following the changes to the SUBJ interface we refine the definition of a generic subject:

```
data ASubj dat =
    ∀ subj. (SUBJ subj, SubjData subj ~ dat) ⇒ ASubj subj
```

which literally says that ASubj dat is *a* subject that publishes data of the type dat.

The operation to link a subject and an observer together can be generically defined as

```
subscribe ::
  (SUBJ subj, OBS obs, SubjData subj ~ ObsData obs) ⇒
  subj → obs → IO ()
subscribe subj obs = do
  subscribe ' subj (AnObs obs)
  subscribed obs (ASubj subj)
```

linking any subject with any observer provided they agree on the published data.

We have thus decoupled the subject and observer interfaces. Although subjects and observers mutually refer to each other, these references are not to concrete types but to interfaces. Hard links are gone, and both the provided and required functionality is abstracted, in the form of SUBJ and OBS classes.

EXERCISE 12. *We called ASubj dat a generic subject. Is it really? Can your make it an instance of SUBJ? Dually, can you write instance OBS (AnObs dat)?*

EXERCISE 13. *Recall that compared to §3, class OBS no longer has the constructor method newObs. How does this exclusion relate to instance OBS (AnObs dat)?*

4.3 Sensors and Displays

As before, the basic publish-subscribe library is used to implement sensors and their displays. Since subjects and observers are decoupled now, so can be sensors and displays. They can be independently implemented and separately compiled. They merely need to agree on the data to publish, i.e., on the observation interface. The module SensorCI defines such an interface:

```
type Label = String
data SensedView = SensedView{
      svLabel :: Label,
      svVal   :: Double}
```

that describes what a Display observed from a Sensor before: its label and the current value.

The module Sensor3 implements sensors. Although the functionality remains as it was in §3, the implementation is quite different now. The module imports the observation interface SensorCI and the publish-subscribe library interface SubObsTr. As before, a sensor has a label and a mutable Double value. That sensor state used to be a part of the subj. Now a subj is a part of the Sensor, which is hence parameterized by the subj implementation.

```
data Sensor subj = Sensor{
    label    :: Label,
    senVal   :: IORef Double,
    subj     :: subj }
```

The construction of sensors is likewise polymorphic over subj, not caring which implementation of subj to use or how it was constructed.

```
newSensor :: subj → Label → IO (Sensor subj)
newSensor s l = do
  v ← newIORef 0
  return $ Sensor{label= l, senVal = v, subj = s}
```

Recall that in §3, a publish-subscribe library implemented a family of subjects subj a parameterized by the concrete subject state. A sensor was a particular instance of the subj family, and as such automatically implements the SUBJ interface and provides methods for publishing and subscribing. Now a sensor contains an implementation of the SUBJ interface rather than being contained it it. Therefore, Sensor subj does not automatically becomes an instance of SUBJ. We have to write such an instance ourselves if a Sensor is to support publishing and subscribing:

```
instance SUBJ subj ⇒ SUBJ (Sensor subj) where
  type SubjData (Sensor subj) = SubjData subj
  publish      = publish      ∘ subj
  subscribe'   = subscribe'   ∘ subj
  unsubscribe' = unsubscribe' ∘ subj
```

The end result is the same as in §3: a Sensor *has* the state (the label and the mutable value) and *is-a* subject. This result is achieved differently however. The above instance demonstrates that subj became a *trait*, which implements only the publish subscription functionality. Anything that contains, or "mixes in", a subj is itself a SUBJ – provided that we write the boilerplate instance like the above.[3] We will address this drawback in §6. (Boilerplate can be eliminated with closed type families just added to GHC.)

We stress that whereas the sensor's value is mutable, the observable value, the one offered in SensedView, is not. A sensor lets observers see its value but not change it. We have easily expressed the access control we could not do before. The following internal function maps the private state to the public view:

```
sensedView :: Sensor subj → IO SensedView
sensedView sen = do
  val ← readIORef ∘ senVal $ sen
  return $ SensedView (label sen) val
```

We now implement the remaining part of the Sensor interface, changing the value of the sensor and notifying all subscribed observers.

```
changeValue :: (SUBJ subj, SubjData subj ~ SensedView) ⇒
                 Sensor subj → Double → IO ()
changeValue sen@Sensor{senVal= vr, subj= s} nv = do
  writeIORef vr nv
  publish sen =≪ sensedView sen
```

The signature is instructive: the subj implementation included in the sensor must support publishing of SensedView data (and subscriptions by observers of that data).

The implementation for sensor observers, displays, has not changed much from §3. The first notable difference is that the implementation is in a separate module Reader3. A display no longer directly accesses private sensor state. All it gets is a public view of a sensor. The second difference is the signature of newDisplay

```
newDisplay ::
  (String → (SensedView → IO ()) → IO (AnObs SensedView)) →
  String → IO (AnObs SensedView)
newDisplay newObs l = newObs l notify
 where
  notify sdata = do
   putStrLn $ unwords ["display",l,"sensor",svLabel sdata,
                        "has_value", show (svVal sdata)]
```

Now newDisplay constructs a generic observer of SensedView using whatever way of constructing an observer: newDisplay is parameterized by the observer constructor, received as the first argument.

EXERCISE 14. *The function changeValue is polymorphic over the sensor implementation. Could we likewise make newDisplay polymorphic over the observer implementation and avoid the existential AnObs? What are the trade-offs?*

We have implemented sensors and their displays in a separately compiled modules, related only by the common interface, SensedView, the public view of a sensor. The private state of the sensor is different and is not directly accessible. Although a sensor is a subject and a display is an observer, they do not depend on the implementation of the publish-subscribe library. They are written entirely in terms of that library interface. To run the tests, however, we have to implement the library.

4.4 Implementing the untangled subjects and observers

The significant changes in the publish-subscribe interface compared to §3 and the untangling of subjects and observers have not affected their implementation that much. The most important difference is that concrete subjects and observers are now in separate, separately compiled, mutually *independent* modules.

Subjects are implemented in Sub3. The only notable change is that Subj is no longer parameterized by the type of the concrete subject state and does not include such state. It is parameterized by the published data.

[3] `mixins.hs` in the accompanied code illustrates in more detail how Scala mixins correspond to Haskell type classes.

```
newtype Subj dat = Subj (IORef [AnObs dat])

instance SUBJ (Subj dat) where
    type SubjData (Subj dat) = dat
    ...
```

The subject implementation type Subj is newtype now. It encapsulates only the subscription list, which contains 'generic observers' AnObs dat. The concrete type of observers may differ (and is hidden anyway); it matters only that they support the OBS interface and accept published data of the type dat. As before, the data constructor Subj is not exported and the realization of the subscription list stays private. In other respects, the code is quite like that in SubObs1, with different type signatures.

The implementation of observers, in Obs3.hs, is even more quite like that of SubObs1. Instead of concrete subject type subj it now refers to 'generic' subjects ASubj dat:

```
data Obs dat =
  Obs{_obsName    :: String,
      _notify     :: dat → IO (),
      obsSubjects :: IORef [ASubj dat]}
```

Even in the implementation of subjects and observers, the hard links are gone, replaced by interface references. The module Obs3 no longer exports Obs, not even the type. Rather, it exports the constructor function, which hides Obs within the generic observer.

```
newObs :: String → (dat → IO ()) → IO (AnObs dat)
```

The test module SRTest3 is hardly different from SRTest2. It imports, now separately, sensors Sensor3 and displays Reader3, and imports the concrete implementation of subjects Sub3 and observers Obs3, again separately. It seems we can easily use a different subject implementation without changing the other components. This is indeed so, as we demonstrate next.

4.5 Extending the subject

The moment of truth for the new design is seeing how much code breaks and has to be duplicated and recompiled when we add the debug printing to the subject library. Luckily, Sub3d – the version of Sub3 with the debug printing in the publish method – can be used with the old implementations of observers, sensors and displays as they were. The test SRTest3d imports the plain Sub3 as S and the debug version Sub3d as Sd, along with the unchanged Sensor3, Reader3 and Obs3. The most interesting test

```
test  = do
  s  ← S.newSubj ≫= \s → newSensor s "sensor1"
  sd ← Sd.newSubj ≫= \s → newSensor s "sensor1d"
  d  ← newDisplay newObs "d1"
  subscribe s  d
  subscribe sd d
```

constructs two sensors s and sd – the former with the plain subjects and the latter with the debug subjects. Recall that newSensor takes as the argument the implementation of subj; any implementation works so long as it supports publishing of SensedView. The test then creates a display, with the Obs3.Obs implementation of observers. The same display can subscribe to the plain sensor and the debug sensor. Although the two sensors have different types, both publish SensedView – which is all the display requires.

The test code showed a simple version of linking required and provided functionality. A Sensor component required an implementation of SUBJ, which newSensor received as the first argument. This 'positional' linking does not scale to component requiring many others. The test code also showed off linking that happened automatically. Recall that subjects and observers are mutually dependent and an implementation of one needs an implementation of the other. The test code did nothing special to tell a subject-sensor which implementation of observer to use. The observer type was determined by the type inference, when type checking the calls to subscribe. §6 gives a larger example of flexible, implicit linking – which does scale.

The new design has fixed all the problems noted at the beginning of §4. Adding the debug printing to the subject implementation no longer forces a new version of observers. Old observers work as before, with the plain and debug version of sensors. Further, a single observer can now subscribe to two differently-typed subjects that have the same type of data to observe. We have attained the characteristic application of the cake pattern: mutually dependent, extensible components with no hard links.

EXERCISE 15. *Is it possible to implement a sensor that supports several observable interfaces?*

EXERCISE 16. *In §3, an observer received the whole subject (rather than a view to its state) and could therefore unsubscribe itself after a notification. Implement a Display that unsubscribes itself after the first notification. How much of the code can be reused?*

EXERCISE 17. *Make a Sensor and Display reusable components themselves by defining their interface.*

4.6 Would a closure have sufficed?

One may wonder if our approach to mutually dependent extensible components is contrived. Can such components be implemented simpler, with a record of closures, with no type classes and existentials? The answer is almost. Closures come tantalizingly close – which explains why the cake pattern is obscure: it only becomes necessary in sufficiently complex cases.

A generic subject ASubj dat of §4.1 is essentially

```
∃ t. {publish :: t → dat → IO (),
         subscribe', unsubscribe' :: t → AnObs dat → IO ()}
```

when the SUBJ dictionary is made explicit. This existential looks exactly as the result of the typed closure-conversion [3] of the record of closures

```
data RSubj dat = RSubj {
  publish      :: dat → IO (),
  subscribe', unsubscribe'  :: RObs dat → IO () }
```

(We have tacitly replaced AnObs with the isomorphic RObs.)

EXERCISE 18. *Constructively prove that ASubj dat is isomorphic to RSubj dat and likewise AnObs dat is isomorphic to RObs dat.*

We now re-implement the running example using only records of closures like RSubj dat and a similar RObs dat, in the simplest Haskell with no type classes, type functions or existentials: see the module SubObsRc in the accompanying code. It starts off easily: the basic subject Sub3 is re-written to produce the closure RSubj dat, see module Sub4. It is even easier to turn the basic observer and the extended one, Display, into RObs dat. As before, these are reusable, swappable components, with no hard links. They are even extensible, to a point, as Display shows. The Sensor is a show-stopper. Recall that a Sensor *is-a* subject, with operations for subscription and publishing. That is, a sensor, in the present approach, must be a value of type RSubj SensedView, a record of three closures. A sensor has an extra operation, changeValue, updating the private sensor state and publishing the updated view. There is no place in RSubj SensedView to put this private data and let changeValue use it, or to put the changeValue closure.

Implementing a sensor thus requires extensible records and row-polymorphism – in essence, the emulation of objects. All these features, and the full OOP, can be emulated in Haskell [2]. Type classes like SUBJ and OBS however emulate just enough of extensible records as needed for component programming, in idiomatic way.

Bringing back SUBJ and OBS lets us complete the example, using Sensor3 with the RSubj implementation of the basic subject, and the RObs implementation of Display.

EXERCISE 19. *Make RSubj dat an instance of SUBJ and RObs dat an instance of OBS.*

The test module SRTest4 differs from the earlier SRTest3 only in imports, of the subjects and observers implementations.

Thus extensible and swappable components, so-called 'dependency injection', can indeed in many cases be implemented in higher-order languages with simple closures[4] Only in sufficiently complex examples, especially like the one below, does the cake pattern becomes indispensable.

5. Proxy: an observer-subject

The extended running example in this section will show the need for the SUBJ and OBS type classes. We see again that SUBJ and OBS behave like traits; an aggregate with an instance of SUBJ itself becomes a subject. The extended example implements a 'proxy' – which is both a subject and an observer. As a subject, it accepts subscriptions from other observers; as an observer, it can be subscribed to a subject and would relay the notifications to proxy's own observers.

There is no easy way to implement such a proxy with the design of §3 or with records of closures of §4.6. In §4.6, a subject was a record RSubj dat and an observer was a distinct record RObs dat. A subject cannot hence be an observer.

EXERCISE 20. *The basic publish-subscribe implementation SubObs2 from §3 likewise had different types for subjects and observers. Can we write another implementation of the SubObsCl, in which subjects and observers have the same type?*

In contrast, with traits of §4, a proxy is not only possible, it is trivial. It is just a pair of a generic subject and a generic observer:

```
data Proxy dat = Proxy{
   prSubj :: ASubj dat,
   prObs  :: AnObs dat }
```

EXERCISE 21. *Could we avoid existentials and parameterize Proxy by implementation types subj and obs, as we did for sensors in §4.3)? Hint: see Ex. 14.*

Recall that §4.3 showed that a sensor is a subject, an instance of SUBJ, because it contains a subject. For the same reason, Proxy is a subject:

```
instance SUBJ (Proxy dat) where
  type SubjData (Proxy dat) = dat
  publish       = publish      ∘ prSubj
  subscribe '   = subscribe'   ∘ prSubj
  unsubscribe'  = unsubscribe' ∘ prSubj
```

Likewise, Proxy dat is an instance of OBS. The Proxy is indeed both a subject and an observer. These two personalities of a proxy are linked when the proxy is constructed:

```
newProxy :: (String → (dat → IO ()) → IO (AnObs dat)) →
            ASubj dat → IO (Proxy dat)
newProxy newObs subj = mfix $ \self → do
  obs ← newObs "proxy" (publish self )
  return (Proxy subj (AnObs obs))
```

Like newDisplay, newProxy takes a constructor of observers; like newSensor it takes a constructed subject. Proxy's notify method publishes the received data to proxy's subscribers, using publish self. The implementation of the notify method therefore needs a reference to the proxy itself. The function

```
mfix :: MonadFix m ⇒ (self → m self ) → m self
```

provides just the needed self-reference for the value constructed as the result of the m self action.

The module ProxyTest demonstrates that proxy indeed can subscribe (to a Sensor) and be subscribed to (by Displays). It truly acts both as a subject and an observer. We have thus seen an example of building a component by aggregation, by composing two smaller components, two smaller traits. Scalable component programming is at hand.

EXERCISE 22. *Define a trait that implements a single-linked list. Combine the two instances of the trait to obtain a double-linked list.*

6. The complete cake

The Scala Cake pattern comes into its own in large applications with numerous mutually dependent components – such as the Scala compiler. A small part of the Scala compiler was the second, full-fledged case study in the paper [6, Sec 4] and the main motivating example in the lecture [5, slides 12–18]. The authors argued that implementing such a set of dependent and extensible components required the combination of functional and object-oriented programming features like those in Scala. In this section we demonstrate that the example in all its complexity is implementable in idiomatic Haskell using the patterns explained §4. A functional programming language alone with higher-rank types and Haskell-like type classes is sufficient.

Scala compiler has to track (term) identifiers such as variables and constants, and their types. Scala is an OO language, and its types are class types, characterized by a set of methods, which are identifiers – hence a mutual dependence between identifiers and types. The example thus is to implement two mutually dependent components, Terms and Types, which extend the common component SymbolTable. Associated with each instance of Terms is an abstract type TermID of identifiers, which can be compared and displayed. Likewise, the Types component provides the abstract type TypeID of class types. Terms should let us associate and find out the TypeID of a given TermID, and Types should provide an operation to find all methods, TermIDs, of a given class type TypeID. Both components need an operation to intern a name (a text string), that is, associate the string with a quickly comparable token. This operation should be factored out in a separate component SymbolTable.

We hence have to implement separately compilable, independently extensible and reusable components. We must maintain encapsulation, to be able to seamlessly replace one implementation by another. The test is adding debug printing to SymbolTable without breaking or even recompiling Terms and Types components. Likewise we should be able to extend the Types component, e.g., with more debug printing. Another requirement, from the real Scala compiler, is to use several instances of Terms and Types within the same program, without mixing them up: attempts to compare TermIDs from different instances of Terms should raise a type error. Different Terms instances should be independent, meaning our implementation should not use any global mutable state.

In the Scala implementation [6], Terms is a class and TermID is a field of that class: classes in Scala are akin to first-class modules of ML, aggregating both types and values. Different instances of Terms will have incompatible TermID types, the consequence of so-called path-dependent typing in Scala. Further, Scala uses so-called selftype annotations to let an implementation of Terms to use the methods of Types and refer to TypeID without mentioning

[4] `http://stackoverflow.com/questions/14327327/dependency-injection-in-haskell-solving-the-task-idiomatically`, see also `cake_currency.hs` in the accompanying code.

any concrete Types implementation. How do we emulate all these features in Haskell?

The Haskell Cake described in §4 gives the answer. The present terms-and-types example is actually an extension of our earlier running example of subjects and observers. New is the common dependency SymbolTable, which does not present any conceptual problem. Emulating Scala path-dependent types is also straightforward, using higher-rank types (see ex2 in the code). In the rest of the section we illustrate the salient new feature: Terms and Types providing not only operations but also abstract types TermID and TypeID to each other, without any hard links. We will also see how Scala self-types look in Haskell. Along the way we demonstrate how to avoid the boilerplate instances for traits seen in §4 and §5 (that is, defining instances proving that each aggregate including, say, a Terms is Terms itself.)

Like before, the interfaces of Terms and Types components are specified as type classes. Although the interfaces are mutually recursive (here and in Scala), they are implemented in non-recursive, separately compiled modules TT/Terms.hs and TT/Types.hs. Recall that associated with each implementation of Terms is a type TermID of term identifiers maintained by that implementation. Type-class–associated types express such a relation between Terms and TermID precisely:

```
type Name = String                          -- concrete name
class Terms repr where
  type Vt repr :: *
  vnew  :: Name → repr → (VT repr, repr)
  ...
```

Here, repr is the type of a particular implementation of Terms interface, and Vt repr is the corresponding type of its identifiers, which we earlier called TermID. The operation vnew returns an identifier with a given name, along with the updated Terms. We have seen this pattern many times in §4. We will modify it however, by separating out the type family Vt from the class Terms. The reason will become clear soon. The Terms component interface thus becomes

```
type family Vt repr :: *          -- corresponds to TermID
type family Tt repr :: *          -- corresponds to TypeID
type MTyp repr = (Tt repr, Tt repr)       -- class method type

class Terms repr vt where
  vnew   :: Name → repr → (vt, repr)
  typeof :: vt → repr → MTyp repr
  ...
```

where Tt repr is TypeID associated with an implementation repr of the Types component. MTyp repr describes the type of Scala methods as a pair whose first component is the class type of the implicit this argument of a method. Terms became a two-parameter class, defining the relation between the implementation repr of Terms and the type of its identifiers. We shall see how this design avoids the boilerplate of traits. The complete interface for terms below reflects the requirement that term identifiers be comparable and showable

```
type TermsCtx repr =
    (Terms repr (Vt repr), Eq (Vt repr), Show (Vt repr))
```

The complete interface of Types is similar:

```
type TypesCtx repr =
    (Types repr (Tt repr), Eq (Tt repr), Show (Tt repr))
```

The implementation of the Terms component (see TT/Terms.hs in the accompanying code) demonstrates defining a trait without the boilerplate and the analogue of Scala selftype annotations in Haskell. We now describe notable parts of the implementation. First, Terms depends on the SymbolTable component providing the type of Symbols, with the following interface

```
class SymbolTable repr where
  type Symbol repr :: *   -- the type of the symbol
  intern :: Name → repr → (Symbol repr, repr)
  ...
type SymbolCtx repr = (SymbolTable repr, Show (Symbol repr),
                       Eq (Symbol repr), Ord (Symbol repr))
```

The implementation of Terms imports the above interface as well as the interface of Types. Again, Terms refers to the required components by their interfaces only. We realize the Terms interface as a map VTable from symbols (representing identifiers) to their method types.

```
data VTable s = VTable (M.Map (Symbol s) (MTyp s))
newtype VN s = VN (Symbol s)
```

VTable s is a concrete type of a Terms instance and VN s is meant as the corresponding TermID type of its identifiers. Since VTable is intended to be incorporated into a larger component, it is parameterized by the type s of that larger component. (One is reminded of two-level types, [9].) We wish to state that VTable realizes the Terms interface; moreover, any aggregate that incorporates a VTable also realizes the Terms interface. The type class Lens makes the notion of 'incorporation' precise:

```
class Lens part whole where
  extract :: whole → part
  update  :: part → whole → whole

extr :: Lens (part repr) repr ⇒ repr → part repr
extr = extract

upd :: Lens (part repr) repr ⇒ part repr → repr → repr
upd = update
```

specifying a relation between a part and the whole. We can now write that anything from which we can extract VTable implements the Terms interface, with the identifier type VN.

```
instance (SymbolCtx repr, TypesCtx repr, Lens (VTable repr) repr) ⇒
    Terms repr (VN repr) where

  -- typeof :: vt → repr → MTyp repr
  typeof (VN key) repr =
      let (VTable m) = extr repr in
      M.findWithDefault (error "typeof:␣can't␣find") key m

  -- vnew :: Name → repr → (vt, repr)
  vnew name repr =
   let (key, r1)    = intern name repr
       VTable m1    = extr r1
   in case M.lookup key m1 of
        Just _  → (VN key, r1)            -- name already seen
        Nothing →
         let (tclass , r2)  = tvar r1
             (tresult , r3) = tvar r2
             VTable m2      = extr r3
         in (VN key,
             upd (VTable (M.insert key (tclass , tresult ) m2)) r3)
```

The instance constraints SymbolCtx and TypesCtx let us use the operations of SymbolMap such as intern and the operations of Types and the type TypeID of class types – without referring to any specific implementation of these components. The dependent components are mentioned entirely by their interfaces. It is instructive compare the above instance with the Scala Cake [6, Slide 17]"

```
trait Symbols { this: Types with Symbols ⇒
  trait Symbol { def tpe: Type }
}
```

(we call Symbols as Terms.) The type annotation on this is the 'selftype annotation', telling that the self type of Symbols should also implement Types. The first line of our Terms instance says exactly the same: anything that incorporates VTable should also implement the Types interface. Haskell does realize selftypes annotations of Scala, in a straightforward way.

EXERCISE 23. *All operations of our Terms also take a TermID. Wouldn't adding a Terms operation debug_print :: repr → String* *with the type not mentioning TermID, break this whole design? How to fix it?*

With the implementation of Terms above and similar for Types and a simple SymbolTable, we can write the example to test all the desired features of the terms and types components from [6, §4]. We have demonstrated that Haskell Cake – type classes with associated types and higher-rank types – reproduce the characteristic features of the Scala Cake.

7. Discussion and Conclusions

We have seen that scalable component programming is possible in Haskell and is relatively straightforward. We can build programs from separately developed and compiled components, with clearly defined interfaces expressed in Haskell rather than in English. The compiler checks that any component implementation conforms to the interface. The components may be mutually dependent and yet are separately extensible. Extending an implementation while maintaining the interface – producing a new version with different algorithms, debugging facilities or new operations – does not require any changes in the dependent components, not even recompilation. Furthermore, two versions of a component may co-exist within the same program. The library versioning problem is thus solvable.

Scalable component programming is a programming pattern, a certain style of writing code. Odersky and Zenger [6] called this pattern Scala Cake and identified its ingredients: abstract type members, explicit selftypes, and symmetric mixin composition. We have demonstrated what these ingredients mean in Haskell terms. It turns out the pattern is realizable in a functional language with no OO. Type classes and higher-rank types are sufficient. In particular, type classes like SUBJ and OBS in §4 emulate just enough of extensible records (objects) as needed for component programming, in idiomatic way.

The recipe for the Haskell Cake is:

- Define an interface for a component in the form of a type class, and place it in a separate module: Type classes emulate ML module signatures. If the interface also calls for abstract types besides operations, use associated type synonyms to introduce those.
- Program a component as a 'trait', to be combined with user data and other components at the final program assembly.
- An implementation of a component should define an instance of the interface type class and export only the corresponding types and their (smart) constructors. (The instances are exported implicitly.)
- Refer to operations and types of other components solely by their interfaces, never by concrete names. Use higher-rank types to implement the references by interfaces.

Together, these ingredients ensure the abstraction over the provided and required functionality of a component. Haskell instance selection mechanism matches up the required and provided features.

Haskell Cake generalizes the well-known pattern of using records of closures to realize abstract data types. Closures indeed suffice for many simple components – that's why we have not come across the Haskell Cake earlier. Closures fall short when components export not only operations but also types and have to be extended with more operations or private data.

We have demonstrated a systematic procedure of replacing hard links (references to concrete types of other components) with soft links, references through interfaces. Informally, if a type T implements an interface Ctx – T is an instance of the type class Ctx – then any reference to T should be replaced with a quantified type variable t subject to the constraint Ctx. There are two ways of carrying this out. In §4.1, the function subscribed referred to an implementation of SUBJ using an existential ASubj. On the other hand, in the signature of changeValue in §4.3, the generic subject was a polymorphic type variable subj with the SUBJ constraint. Likewise, the proxy data type in §5 aggregated two existentials AnObs and ASubj.

EXERCISE 24. *Implement Proxy in §5 with universals instead.*

Using existentials and universals for soft links parallels to universals and existentials as two alternatives to represent abstract data types described in [4, §3.8]. Mitchell and Plotkin also briefly discuss the trade-offs. We should add that universals let us implement binary methods but cause the proliferation of type parameters to data types and operations. Existentials have the opposite trade-off; also, a type class dictionary carried in each value of the existential type may be a significant overhead.

Developing traits requires certain amount of boilerplate: if a type class SUBJ specifies the interface for a component (trait) and Subj1 is one of its implementation, to prove that any value that incorporates Subj1 is also an implementation of SUBJ we have to write an instance of SUBJ. That instance is boilerplate and could be mechanically derived, e.g., using Template Haskell. Closed type families just added to GHC could let us state once and for all that anything that incorporates a Subj1 is a SUBJ itself, thus avoiding the boilerplate. §6 showed a different way around the boilerplate, requiring more type class parameters. We need more experience with developing traits to clarify the trade-offs of these approaches.

Abstraction has cost. Implicit linking of components adds a level of indirection – via the virtual method dispatch in Scala or the dictionary method look-up in Haskell. Luckily for Scala, which compiles to JVM, just-in-time compilers well optimize virtual method invocations. Higher-rank types seem to make it difficult to eliminate the overhead of type class method calls, although a whole-program analysis may help: an implementation of concepts in C++ suffers from the same indirection overhead, which can be eliminated however by static analysis [10].

In conclusion, we can write scalable component in Haskell right now. We should really try following the Haskell Cake in various projects, despite the fair amount of boilerplate in some cases. Only then we gather experience that will suggest what sort of sugar has to be added to Haskell to make the cake sweeter.

Acknowledgments

I am greatly thankful to Johan Tibell for the encouragement and many helpful conversations.

Answers to selected exercises is in the text itself; see also the accompanying code.

References

[1] M. M. T. Chakravarty, G. Keller, S. L. Peyton Jones, and S. Marlow. Associated types with class. In J. Palsberg and M. Abadi, editors, *POPL '05: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 1–13, New York, 2005. ACM Press. ISBN 1-58113-830-X.

[2] O. Kiselyov and R. Laemmel. Haskell's overlooked object system. *CoRR*, abs/cs/0509027, 2005.

[3] Y. Minamide, J. G. Morrisett, and R. Harper. Typed closure conversion. In *POPL '96: Conference Record of the Annual ACM Symposium on Principles of Programming Languages*, pages 271–283, New York, 1996. ACM Press. ISBN 0-89791-769-3. URL `http://www.cs.cmu.edu/~rwh/papers/closures/popl96.ps`.

[4] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[5] M. Odersky. Objects and modules – two sides of the same coin? Milner Symposium, 16 Apr. 2012. URL `\url{http://events.inf.ed.ac.uk/Milner2012/slides/Odersky/Odersky.pdf}`.

[6] M. Odersky and M. Zenger. Scalable component abstractions. In *OOPSLA*, pages 41–57. ACM, 2005. doi: 10.1145/1094811.1094815. URL `\url{http://lampwww.epfl.ch/~odersky/papers/ScalableComponent.pdf}`.

[7] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12): 1053–1058, 1972. URL `http://www.cs.umd.edu/class/spring2003/cmsc838p/Design/criteria.pdf`.

[8] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing '83: Proceedings of the IFIP 9th World Computer Congress*, pages 513–523, Amsterdam, 1983. Elsevier Science. URL `ftp://ftp.cs.cmu.edu/user/jcr/typesabpara.pdf`.

[9] T. Sheard and E. Pašalić. Two-level types and parameterized modules. *Journal of Functional Programming*, 14(5):547–587, Sept. 2004.

[10] J. G. Siek. The C++0x "concepts" effort. In J. Gibbons, editor, *SSGIP*, volume 7470 of *LNCS*, pages 175–216. Springer, 2012. ISBN 978-3-642-32201-3. URL `http://dx.doi.org/10.1007/978-3-642-32202-0`.