Modularization of Algorithms on Complex Data Structures

An Encoding of Typesafe Extensible Functional Objects

Jonathan Immanuel Brachthäuser

Fachbereich Mathematik und Informatik
Philipps-Universität Marburg

A thesis submitted for the degree of

Master of Science

Supervisors:
Tillmann Rendel
Prof. Dr. Klaus Ostermann

September 29, 2014

Integrity Statement

Herein I declare that this Master Thesis was created entirely by myself. I only used the sources and tools specifically stated in this document. Thoughts used, either by meaning or quoted, were marked as such.

Marburg, September 29, 2014 Jonathan Immanuel Brachthäuser

Abstract

Modularization is the process of decomposing a system into multiple components that then can be developed in isolation. It thereby enables separate compilation and facilitates modular reasoning and understanding of programs.

Two aspects of modularization can be identified. The first, and most prevalent aspect is to decompose a system statically according to its responsibilities. In this thesis we take another view on modularization: The decomposition of a system in terms of temporal aspects.

Using objects to represent modules, the temporal aspect of modularization is captured by *dynamic specialization*. Over the course of the program execution, objects can be augmented with additional operations step-by-step. The incremental acquisition of information and refinement of components very naturally matches how humans acquire knowledge and refine their mental model.

Dynamic runtime adaptation and refinement of objects is not available in most statically typed, class-based programming languages. An analysis of existing solutions to this problem reveals a few approaches that seem suitable for our purposes: The language *gbeta*, the language extension of *generic wrappers* and the *calculus of incomplete objects*. However, all three solutions lack a seamless integration into existing software development environments.

Encodings allow expressing language features within a (different) host language, while reusing existing infrastructure and facilitating reuse. In particular, no custom compiler or preprocessor is necessary.

We propose an encoding of "typesafe extensible functional objects" to solve the problem of temporal modularization and to support the programming paradigm of dynamic specialization. A prototype of the encoding is embedded into Scala, a statically typed, multi paradigm language that supports both functional as well a object-oriented programming.

The solution presented in this thesis is based on a standard coalgebraic encoding of objects, where interfaces are encoded as endofunctors and classes are encoded as coalgebras over these endofunctors. Objects are then represented by the terminal coalgebra, which can be obtained by unfolding a coalgebra with an initial state.

Building on the standard encoding, we define composition of coalgebras modeling static composition of traits and add extension points to the infinite tree of observations (the terminal coalgebra). The added extension points enable later refinement of the object implementation in terms of composing the original coalgebraic implementation with the provided extension. Our encoding is typesafe. It reuses Scala's type system to express dependencies between different extensions to assure that all dependencies are fulfilled at runtime and method calls can be resolved. Extensions in our encoding are transparent. An extended object can be used everywhere the original object could have been used. In consequence, objects can be extended with multiple extensions, aggregating all features described by the individual extensions. Our encoding supports late binding. The self-reference is always bound to the receiver of a method call, enabling extensions to override methods and refine the behavior of an object.

Extensions for referencing the base object and selective open recursion have been developed on top of the core encoding. As a proof of concept, our encoding has been evaluated by the analysis of three use case examples.

Zusammenfassung

Modularisierung bezeichnet den Prozess, bei dem ein System in mehrere Komponenten aufgeteilt wird, sodass diese getrennt voneinander entwickelt werden können. Dadurch ist es möglich, Komponenten separat zu kompilieren, unabhängig voneinander zu testen und sie zu verstehen, ohne dabei notwendigerweise die anderen Komponenten kennen zu müssen.

Es können zwei Aspekte von Modularität betrachtet werden. Der erste Aspekt betrifft das statische Aufteilen von Systemen in Module anhand ihrer Zuständigkeit. Diese Arbeit fokussiert allerdings den zweiten und seltener berücksichtigten Aspekt von Modularität zu Grunde: Die Aufteilung eines Systems in Komponenten anhand zeitlicher Kriterien.

Werden Module durch Objekte repräsent, so kann der zeitliche Aspekt der Modularisierung durch sogenannte *Dynamic Specialization* ausgedrückt werden. Dynamic Specialization erlaubt es, während der Laufzeit eines Programmes Objekte schrittweise mit neuen Operationen auszustatten. Das inkrementelle Sammeln von Information ähnelt hierbei dem Prozess menschlicher Wahrnehmung und Bildung mentaler Modelle.

Dynamische Anpassung und Erweiterung von Objekten ist in den meisten klassenbasierten und statisch getypten Programmiersprachen nicht verfügbar. Eine Analyse von Arbeiten zu diesem Problem zeigt, dass einige der beschriebenen Lösungsansätze gut geeignet scheinen. Hierzu zählen die Sprache *gbeta*, die Spracherweiterung *Generic Wrappers*, sowie der *Calculus of Incomplete Objects* (Kalkül unvollständiger Objekte). Alle drei Lösungen können leider nicht direkt in bestehende Softwareentwicklungsprozesse eingebunden werden, da sie spezielle Compiler benötigen und damit nicht kompatibel zu existierenden Bibliotheken und Entwicklungsumgebungen sind.

Encodings erlauben es, ein Sprachkonstrukt in einer (anderen) *Host-Sprache* auszudrücken, welche dieses Element möglicherweise nicht unterstützt. Das Encoding verwendet ausschließlich die Host-Sprache, um die Semantik des Sprachkonstruktes auszudrücken. Dadurch ist es möglich, weiterhin existierende Softwareentwicklungsprozesse und Bibliotheken zu verwenden.

Im Rahmen dieser Thesis wird ein Encoding von typsicheren, erweiterbaren, funktionalen Objekten vorgestellt, um das Problem der Modularisierung anhand zeitlicher Aspekte zu lösen und Dynamic Specialization zu unterstützen. Eine prototypische Implementierung wurde in der Programmiersprache Scala vorgenommen. Scala ist eine statisch getypte, Multi-Paradigmen Sprache, die sowohl funktionale als auch Objektorientiere Programmierung erlaubt.

Die hier vorgestellte Lösung basiert auf einem aus der Literatur bekannten koalgebraischen Encoding von Objekten. Hierbei werden Schnittstellen als Endofunktoren und Klassen als Koalgebren über diese Funktoren dargestellt. Objekte entsprechen dann der terminalen Koalgebra, dem unendlichen Baum möglicher Beobachtungen auf einem Objekt.

Auf dieser Interpretation von Objekten aufbauend, wird in dieser Arbeit die statische Komposition von Klassen durch Komposition von Koalgebren modelliert. Desweiteren wird der unendliche Baum von Beobachtungen auf jeder Ebene mit Erweiterungspunkten versehen, die es ermöglichen, die Implementierung eines Objekts in Form der ursprünglichen Koalgebra durch eine spezialisierte Koalgebra auszutauschen

und hiermit das Objekt um neue Funktionalität zu erweitern.

Das Encoding ist typsicher, denn Abhängigkeiten zwischen verschiedenen Erweiterungen werden im Scala Typsystem ausgedrückt. Dadurch wird sicher gestellt, dass diese zur Laufzeit erfüllt sind und Methodenaufrufe korrekt aufgelöst werden können. Erweiterungen im Encoding sind "transparent" für den Nutzer, denn erweiterte Objekte können überall dort verwendet werden, wo das ursprüngliche Objekt erwartet wurde. Insbesondere können Objekte mehrfach erweitert werden und aggregieren hierbei sämtliche Operationen, die durch die verschiedenen Erweiterungen beigetragen werden. Das Encoding unterstützt späte Bindung, denn Referenzen zum Objekt selbst werden immer an die letzte Erweiterung des Objekts gebunden. Dadurch können Erweiterungen Methoden überschreiben und das Verhalten eines Objektes spezialisieren.

Auf Basis des Encodings, wurden zwei Spracherweiterungen entwickelt. Die erste ermöglicht es das ursprüngliche Basisobjekt innerhalb einer Erweiterung zu referenzieren. Die zweite erlaubt es auszuwählen, ob ein Methodenaufruf spät gebunden werden soll, oder nicht. Das Encoding wurde anhand von drei Beispielprogrammen evaluiert.

Acknowledgments

I would like to thank Bruno Oliveira for his valuable feedback and inspiring discussions on my research at ICFP'14. I am particularly grateful for the assistance given by Prof. Ostermann as well as ACM and Microsoft Research that allowed me to present my work at the ICFP'14 Student Research Competition in Gothenburg. This thesis would be impossible without the inspiring discussions with my supervisors Prof. Ostermann and Tillmann Rendel. My grateful thanks are also extended to Paolo Giarrusso who spent countless hours explaining me the inner workings of Scala and who provided much appreciated feedback on early versions of this thesis. I would also like to thank Nada Amin and Marlen Müeller for their constructive comments on early versions of this thesis. Finally, I would like to express my deep gratitude to Tillmann Rendel who, with his enthusiastic encouragement, has been a really great mentor.

Contents

List of Figures

Chapter 1

Introduction

It is difficult to implement algorithms on complex data structures. This is especially the case when those data structures are part of an evolving software system with possibly multiple parties involved, each of them independently working on their share of the large scale system. Modularization deals with this problem, often by introducing abstraction barriers like interfaces or contracts (Meyer, 1992) and thereby performing information hiding (Parnas, 1972). Stable interfaces protect user code from internal implementation changes by explicitly articulating guarantees. At the same time it protects the implementor of the interfaces from undesired assumptions about the inner workings made by the client. It thus encourages an evolution of the implementation without having to concern with breaking user code.

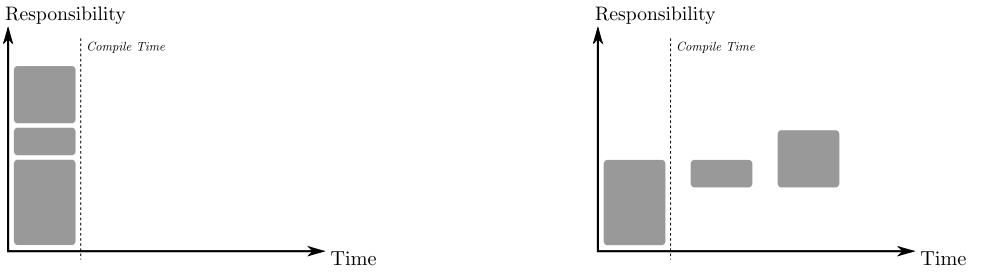
Software components are modules which are developed by third-parties and possibly only distributed as closed source in binary form. Component-oriented software development (Nierstrasz et al., 1992) aims at creating reusable components that are then adapted and combined to yield complete software systems. One important part of working with software components is hence to add extensions in order to customize the components to the use-site context.

This gives rise to another notion of modularity – one that is concerned with the dynamic evolution of a system. With the term *temporal modularity* we try to capture the conceptual nature of modularity that we are concerned with in this thesis. Hence, temporal modularity describes decomposing the behavior of a system into components that can be applied at runtime depending on the state of the program. Figure 1.1 illustrates the difference between the two notions of modularity. While with the usual notion of modularity decomposition takes place in its entirety before compile time (Figure 1.1a), temporal modularity partitions the software artifact according to runtime criteria (Figure 1.1b). We neither claim that one aspect is more important than the other, nor that one can subsume the other. However, we believe that supporting emphboth aspects can facilitate implementing modularized algorithms on complex data structures.

1.1 Extensibility and the Expression Problem

Decomposing algorithms statically while preserving extensibility is already difficult. One important problem in this context is the *expression problem* (Wadler, 1998). This section briefly introduces the expression problem, as well as one particular solution that had a big influence on our approach: object algebras (Oliveira and Cook, 2012).

When modularly defining algorithms on data structures, we might think of two different dimensions to decompose the modules accordingly. Both dimensions offer different kinds of extensibility. The first dimension chooses the structure of the data as the dominant decomposition criterion. This allows adding new *variants* of data types later. The second dimension focuses on the algorithms which are defined. This allows adding new *operations* later, that is, defining new functions over the data type in functional programming or adding methods to the corresponding classes in object



(a) Structuring software into components that separately can be compiled, but are composed at link-time.

(b) Structuring software into components that selectively can be defined and composed at runtime.

Figure 1.1: Two different notions of modularity.

oriented programming. Supporting both kinds of extensibility is well-known to be a difficult problem (Wadler, 1998).

- Class-based object-oriented programming languages offer good support for one dimension of extensibility: Adding new variants to a class hierarchy by subclassing a common interface. At the same time it is difficult to add methods to all variants of class hierarchy without modifying existing code.
- Functional programming languages offer good support for the other dimension of extensibility: Adding new functions defined on a data type by pattern matching on the variants. At the same time it is difficult to add new variants to a data type which also requires adding new cases to the function definitions.

Oliveira and Cook (2012) introduced object algebras as a program-structuring technique that allows modularizing algorithms on data structures while keeping the definition of data structures open to support extensibility. The notion of an object algebra serves as a first-class representation of an algorithmic component and thus facilitates both extensibility and modularity. The core idea is based on the isomorphism

$$(S + T) \to A \approx (S \to A) \times (T \to A)$$

that is, a function defined on a sum-of-products (where S and T are variants) is isomorphic to a product containing one function handling each variant. Using this isomorphism Oliveira and Cook bring the functional style of defining programs to object oriented languages. The cases of pattern matching ($S+T$) translate to methods of the object algebra ($S \to A$ and $T \to A$). A program on a data type can be defined by implementing the interface of the object algebra corresponding to that data type. Since object algebras are classes, the advantages of object oriented programming can be used to also add new variants hence solving the expression problem.

1.2 Dynamic Extensibility

Object algebras allow to modularly define algorithms by decomposing finite data structures. Here "modularly" refers to the way object algebras are defined as separate components according to the features they implement (Oliveira et al., 2013). Once an object algebra is constructed, it can be applied to a given data structure in order to perform the necessary computation.

However, traversing trees of data is not the only program structuring technique that is used for real software system and complex algorithms. Where the definition

of functions over data structures is a technique that originates in functional programming, *objects* are an alternative way of structuring programs widely used in object-oriented programming.

Objects encapsulate a hidden state together with the description of behavior. The behavior can be observed by invoking methods on the object, possibly leading to a change of the internal state. Objects typically outlive a single method call. This is different with functions defined on data structures. They typically describe stateless computation for a given static data structure.

We have seen how object algebras can be used to define algorithms modularly. What does it mean to modularly define objects? (Ernst, 1999) defines *dynamic specialization* as augmenting objects step-by-step with additional operations over the course of a program execution. The incremental acquisition of information and refinement of components very naturally matches how humans acquire knowledge and refine their mental model. With every part of new information, the behavior of an object can be adapted by adding new methods or redefining existing ones.

In this thesis we take the standpoint that *dynamic specialization*, aiding temporal modularization, is the dual to (statically) modularizing algorithms by decomposing them into traversal-components.

While the previous motivation was rather theoretical, a few examples show why dynamic object specialization is actually useful in practice. Dynamic specialization can be used for:

- customization of objects according to language localization only known at runtime,
- adding methods for printing and tracing in order to facilitate debugging,
- creation of mock objects in test-driven-development,
- incremental construction of objects performed by modularized builders,
- "monkey-patching" behavior of library objects to address application specific requirements and
- annotating objects with additional information, acquired after object creation.

In summary, dynamic specialization of objects allows *modularization* according to temporal decomposition criteria, *dynamic adaption* to the computational context as well as "just-in-time" *incremental specification* of objects.

Since dynamic specialization seems to be very useful, the following question arises:

> *Can dynamic specialization of objects be embedded into a statically typed programming language?*

1.3 Our Approach

As it turns out, there is a close correspondence between the behavior of objects and the definition of algorithms on data structures. Where *folding an algebra* over abstract data types represents computation performed on data structures, *unfolding a coalgebra* with an initial state corresponds to object creation (Jacobs, 1995; Lämmel and Rypacek, 2008). We thus might need to rephrase the above question slightly to: "Can *unfold*, that is, object instantiation, be incrementalized to create a modular definition of codata"?

In order to answer this question, we developed *obj*.**extend**, an encoding of typesafe extensible functional objects based on coalgebras. For this purpose, the approach of

translating pattern matching for function definition into object algebras is dualized. In our approach we translate mixins that define objects into coalgebras, which are functions.

For the implementation of *obj*.**extend** we use Scala (Odersky and Zenger, 2005), a statically typed programming language with both support for functional programming and object oriented programming idioms. Scala has no support for dynamic specialization and thus does not allow to extend the interface of already constructed objects. This limitation is quite common in the field of statically typed, class-based programming languages. Other prominent examples include Java, C++, Objective C and Eiffel. However, with *traits*, Scala has advanced support for static modularization of classes. Traits can be thought of as interfaces that are also allowed to contain implementation. To construct objects, multiple traits can be statically composed and afterwards instantiated. However, to the current day it is neither possible

a) to mix-in traits not known at compile time of the defining module nor

b) to mix-in traits into already initialized objects.

This thesis sets out to bring dynamic object specialization as a library to the Scala language in order to allow the dynamic modularization of algorithms on complex data structures. For this purpose an encoding of functional objects is developed that is applicable in multi paradigm languages like Scala. It can be used as a library and does not require any change to the existing compiler. While many advanced Scala features are used to syntactically ease the use of the encoding, the only crucial features that are not frequently found in other languages are intersection types, higher-kinded types and variance annotations. In summary, the contributions of this thesis are:

- An encoding of functional objects that also supports dynamic specialization (Section 3.4). For this purpose we first adapt the standard encoding (Section 3.1) to also emulate features that are already present in Scala such as:
  1. self-type annotations and private state (Section 3.3),
  2. subtyping in presence of an isorecursive fixed point construction (Section 3.4.3),
  3. references to the *base* similar to **super** calls (Section 4.1).
- An extension based on the core-encoding that allows to express use-site *selective open recursion* (Section 4.2).
- Translation of several small use case examples to our encoding (Section 5.1),
- Phrasing object extensibility and dynamic specialization coalgebraically as a combination of a modified *unfold* and composition on coalgebras (Section 3.4).
- A delegation based composition of type constructors[1]. (Section 3.2).
- Automatic generation of functor instances in Scala (Section 3.2).

The source code, containing the prototype implementation of our encoding, as well as the use case examples is available under `http://obj.extend.b-studios.de`. The implementation is a proof of concept and thus not yet ready for production use.

We hope that our encoding meaningfully complements object algebras and that the two techniques can be used together to unlock extensibility not possible before.

[1] An early version of the composition operator *mix* for types of kind $*$ has already been used in our earlier work on (Rendel et al., 2014) and thus cannot fully be counted as contribution to this thesis

Chapter 2

Motivation and Context

This chapter provides motivation for *dynamic specialization* by example. The example then is used to motivate a set of desired requirements that should be met by solution candidates. A summary of related work evaluates individual solutions with respect to the requirements, focusing on the context of dynamic specialization.

2.1 Dynamic Specialization: a Real World Example

Imagine we are trying to write an object relational database mapper (ORM) for a customer. We have just read the book "Patterns of Enterprise Application Architecture" by Fowler (2002), and the Active Record design pattern caught our attention: "An object that wraps a row in a database table [...], encapsulates the database access, and adds domain logic on that data" (Fowler, 2002, p. 160).

Soon after, we start working on an API that represents the rows of the database as objects. Very quickly, we notice that our implementation performs really badly. As a consequence, we reify our database queries and also represent them as objects. On the one hand this allows to perform optimization of the queries before executing them, on the other hand the execution of queries can be delayed until the results are actually used.

Everything is packed up in a nice domain specific language where queries look like

$$\text{BusinessObject}.\mathit{all}.\mathit{filter}\ \{...\}.\mathit{sortByName}$$

Proudly, we present the new library to our customer who immediately comes up with a new requirement: She would like to be able to define own custom query-methods like $\mathit{query}.\mathit{medianValue}\ (\mathit{attr})$ – and additionally they should be stateful to allow advanced caching strategies. Inclined to serve our customer we agree to consider the request, but soon we realize that the requirements are not trivial.

The essence of the problem is, that we want to enable the client to freely add functionality to library objects and at the same time do not want to expose details on how the objects are constructed to perform information hiding (Parnas, 1972). The responsibility of constructing objects is on the side of the framework. We could add some configuration mechanisms, maybe by using the builder pattern. However, we cannot anticipate all possible extensions our client and other future clients will request. So how can we dynamically augment objects in retrospect?

Dynamically typed languages like Ruby or JavaScript feature a liberal model of extensibility where objects always can be augmented, building the basis for programming techniques such as *monkey patching* – changing program behavior at runtime. Developers from the Ruby-on-Rails team encountered similar requirements as those issued by our fictional customer. They were able to address them quite easily, using Ruby's support for dynamic inheritance, to offer the method *extending* in their API[1]. Calling $\mathit{query}.\mathit{extending}\ (\mathit{Custom})$ on a query object will dynamically mix-in

[1] `http://apidock.com/rails/ActiveRecord/QueryMethods/extending`

the module *Custom* into the query object, augmenting its interface by the methods define in *Custom*.

With the flexibility comes additional complexity of the resulting programs. It is not visible to the user of a library how objects are created and how they might be modified by other users during their lifetime. This increases the difficulty to reason about program behavior, resulting in the bad reputation of some features of dynamic languages. This is especially a problem since the above mentioned languages lack a static type system. However, enabling specialization of objects at runtime in a language with static typing can arguably give more guarantees. For instance it could prevent erroneous access of non existing methods, also known as *NoMethodError* in Ruby.

Despite the runtime behavior that is sometimes hard to comprehend and especially hard to foresee, many researchers have been attracted by the flexibility and expressiveness of dynamic languages. The remainder of this Chapter will revisit existing solution candidates. The selection of related work was guided by the following list of requirements which draws much inspiration from (Büchi and Weck, 2000).

1. Extensibility at Runtime. In our example above, the objects representing database queries are instantiated by the framework, not the client. The client only obtains instances by the use of framework methods and hence cannot immediately influence the construction of objects. While some configuration options might be anticipated, custom extensions created after the framework certainly cannot. Thus we demand extensibility of objects, *after* they have been created.

2. Transparency. The actual type of an extended object should be both *(a)* a subtype of the *actual type* of the object (the base) and *(b)* a subtype of the extension that has been applied. For example, an extension that has been designed to be applied to instances of a class *Query*, should also be applicable if the actual type of a query object is a subtype of *Query*, like $GroupingQuery <: Query$. In particular, after extension, we still want to be able to call methods available on grouping queries such as *getGroupingProperty*. At the same time also the newly added methods implemented by the extension should be available. The extension thus should not hide the actual type of the base but behave *transparently*[2].

3. Late binding. Extensions should be able to override methods while preserving late binding. Calls to methods should always be resolved with respect to the dynamic receiver at runtime.

Let us assume we want to add a caching mechanism for the execution of queries in retrospect. Every time the result of a query is requested using the method *get*, we first want to look up the result in our cache, before possibly executing the query to update our cache in the case of a cache miss. The caching should also be applied for requests that are performed from within other method implementations.

4. Permanency. Extensions added to an object should be persistent. That is, after an object has been extended, all subsequent usages of the object should refer to the extended interface. This is important in particular, since extensions itself might carry state such as the above mentioned cache.

[2] What we call "transparency" is originally split into two separate requirements (Büchi and Weck, 2000). The authors use the term "genericity" to describe the fact that extensions can also be applied to subtypes of the specified base-type and "transparency" to require that the result of applying an extension is an element of the actual type of the base and of extension type. We drop "genericity", since it is implied by "transparency".

5. Integration. The integration into existing software systems should be seamless. This requirement is twofold: On the one hand, the transition costs to a possible solution should be kept low. It should be possible to reuse existing infrastructure such as development environments and build pipelines. On the other hand, it should be possible to extend objects instantiated from existing legacy classes without having to modify the source or the binary code of these classes.

2.2 Current State of the Art

The desire to dynamically adapt the runtime behavior of objects in statically typed, class-based programming languages has a long standing history. Multiple decades of research spawned a variety of solutions. From design patterns over language extension proposals to specialized calculi. In this section we will discuss some of the solutions in the light of dynamic specialization, highlighting the different benefits and disadvantages. While there is a large body of research on dynamically typed programming languages, we will focus on statically typed approaches, only.

2.2.1 Dynamic Specialization and gbeta

The term "dynamic specialization" was coined by Erik Ernst in his PhD thesis on gbeta (Ernst, 1999, Ch. 7.3). He describes dynamic specialization as a very natural refinement process, driven by incremental knowledge acquisition which he calls *discovery*. A programming language should thus be able to reflect the procedural refinement of objects.

As an example, we can inspect how a network package is processed by the several layers of an TCP/IP stack. Arriving on the link layer a package might be represented as a *Frame* class that has fields for the header and footer as well as the contained frame data. The package object then is passed on to the next layer "internet" which analyses the frame data to extract the IP header information like the IP address and the corresponding IP data. Since the internet protocol is the basis for routing, after this step, the package object might be refined to be also of the class *Routeable*. Likewise, the transport layer might analyze the object to further specialize it to be *Transportable*. Finally, the application layer might pass the package object to the module that handles the corresponding high level protocol (for instance HTTP or FTP), possibly again specializing the package object.

This example shows how a very simple package object evolves from an instance of *Frame* to an instance of the specialized type *Frame* & *Routeable* & *Transportable* & *HttpPackage*. While the first processing layers of the stack are fairly static and thus could be foreseen in the design of a software system, the processing in the application layer does dynamically depend on the package content.

As a generalization of the BETA language (Madsen et al.), gbeta (Ernst, 1999) introduces a few features that are unusual for a statically typed language. Classes can be created and merged dynamically at runtime. Further, it is possible to inherit from dynamically created classes by inheriting from *class variables*. All of this is possible, since classes are first-class citizens in gbeta. Lastly, and most important for this thesis, in gbeta an object can be dynamically specialized with a class to also be an instance of that given class.

Let us translate the above TCP/IP example to gbeta. Some example classes for *Frame*, *Routeable*, *Transportable* and *HttpPackage* can be defined as in Figure 2.1. The unusual syntax %(...) {...} defines a *pattern*. In gbeta classes and methods are unified under the concept of patterns. Patterns can specify input and output within

```
Frame: %{frameBody: string};
Routeable: %{
  routeTo: %(str: string) {"routing to " + str | stdio};
};
Transportable: %{protocol: %(| "TCP")};
HttpPackage: %{
  httpHeaders: %(| "GET / HTTP/1.1\nHost: example.org\n\n");
};
```

Figure 2.1: Example of a dummy TCP/IP stack implementation in gbeta.

the pair of parenthesis, separated with a pipe-character. Members and computation can be defined within the curly braces. If no output is present, the pipe can be omitted. If neither input or output is necessary, the parenthesis can also be omitted as a whole.

The class *Frame* has one uninitialized member *frameBody* of type *string*. *Routeable* contains the method *routeTo* which takes a string and prints the result to *stdio*. The classes *Transportable* and *HttpPackage* each define one constant method that always returns the same string.

Given *package*, a reference to an object of type *Frame* we can print the contents of the *frameBody* property to the standard output.

```
package.frameBody + "\n" | stdio;
```

The pipe character here represents message passing. Hence, the result of evaluating *package.frameBody* + `"\n"` is passed as input arguments to *stdio*. After having analyzed the frame body, we might want to specialize *package* to also be of type *Routeable*. In gbeta this is straightforward:

```
Routeable# | package#;
```

This refines the pattern of the object *package* with the pattern *Routeable* to result in the merge *Frame* & *Routeable*. We can match on the runtime pattern of an object to use the newly gained functionality:

```
case it: package do {
   ? Routeable: "127.0.0.1" | it.routeTo;
};
```

This results in "`routing to 127.0.0.1`" being printed on the console.

The same refinement can of course also be applied with the classes *Transportable* or *HttpPackage*. Since the object itself is refined, the object identity is preserved and all references to *package* can make use of the extended functionality. The extension is permanent.

Since our definition of dynamic specialization origins from (Ernst, 1999) it is not surprising that gbeta matches almost all of our requirements specified above. Patterns are first class and can be merged into objects at runtime. Late binding is supported by means of virtual attributes (Ernst, 1999, Ch. 4.2). gbeta only fails the requirement of integration into existing systems. As a full general purpose programming language, gbeta requires its own compiler and development infrastructure.

2.2.2 Static Extensibility

Next, we will revisit some solutions that *statically* allow the decomposition of classes into separate modules. In each of the solutions, the modules can be type checked separately, supporting modular reasoning. Since the approaches are static, non of them supports extensibility at runtime. Nevertheless, considering these technologies is interesting since it shows the difference between static and dynamic modularization.

Static Mixin Composition. Scala offers static mixin composition (Odersky and Zenger, 2005; Bracha and Cook, 1990) of multiple separately defined *traits* to yield what is represented by *intersection types* on the type level. Traits are allowed to contain abstract as well as concrete members which automatically complement each other when being mixed together. In contrast to abstract members, *self-type annotations* allow a more course grained nominal specification of interfaces that need to be mixed in, before an object can be instantiated from the trait. The clause **trait** A **extends** B, explicitly requires that B will follow after A in the chain of base classes. Thus a call to **super** within A will eventually lead to the invokation of the corresponding method in B. Specifying the same using a self-type annotation **trait** $A\,\{\,self : B \Rightarrow\}$ only assures that B will be mixed in before the object can be initialized. It does not articulate assumption about how A and B relate in the chain of base classes. Using self-type annotations it is thus possible to express circular dependencies, since any order of the mutual dependent traits will meet the requirements.

In order to avoid problems usually associated with multiple inheritance the Scala compiler performs linearization by topologically sorting all the collected super classes according to the order in which they are specified. After linearization, every call to **super** is deterministic and hence does not lead to ambiguities.

With mixins we can decompose the monolithic definition of a class C into multiple traits, say A and B. Objects can be immediately be instantiated from both component traits by

val *obj* = **new** A **with** $B\,\{\,\}$

This creates the intersection type of A and B, performs linearization of the super classes and checks whether all requirements on the self-type are met. However, all involved traits always have to be known at compile time. In our example A and B both have to be statically known and for instance cannot be bound by type parameters. Another limitation is that traits cannot be mixed into existing objects. Hence, as mentioned above, our first and most important requirement of runtime extensibility is not met.

Open Classes. Another line of static extensibility is represented by *open classes* (Millstein and Chambers, 1999; Clifton et al., 2000) and *extension methods* as found in the C♯ programming language. Both techniques allow the retroactive addition of methods to existing classes, without having to change the original source. That is, classes that are defined in third party modules can be extended with additional methods at compile time. An example from (Clifton et al., 2000) illustrates the definition of a method *rotate* on the class *Shape* which is already defined in another compilation unit.

public Shape Shape.*rotate* (**float** *a*) {...}

Compared to Scala traits, which are closed after definition, open classes and extension methods represent an improvement regarding the extensibility. However, with

both techniques it is not possible to dynamically extend individual objects with new functionality. In addition, extension methods as implemented in C♯ do not support late binding and thus also fail to meet this requirement.

2.2.3 Design Patterns

Using composition to overcome the limitations of (static) inheritance is a well known refactoring technique (Johnson and Foote, 1988; Fowler, 1997; Kegel and Steimann, 2008) and the core idea of many design patterns (Gamma et al., 1994). In this subsection we highlight two design patterns that are specifically targeting dynamic specialization.

Decorator Pattern. The decorator pattern is a design pattern (Gamma et al., 1994) that allows dynamic extension of objects with decorators. For this purpose, the decorating class is a subclass of the *base class* (the class of the object being decorated), adding new methods to the interface, possibly overriding existing methods and forwarding all other methods to the base object as default implementation. Decorators are often set up in a class hierarchy where all implementations and decorators inherit from an abstract interface, allowing to arbitrarily nest several decorators on top of implementing classes.

While supporting extensibility at runtime, the decorator pattern suffers from multiple problems. First of all, independently developed extensions cannot be composed, since methods added to the interface of one decorator cannot be anticipated by another decorator. In consequence, only the interface of the last decorator is visible to the client. Assuming two decorators *QueryLogger* and *QueryCache*, applying both decorators and invoking methods, that are only defined on the inner decorator will fail:

new QueryCache (**new** QueryLogger (*query*)).*logMessages* // *Type Error!*

This example illustrates, that the decorator pattern does not account for transparency as required earlier in this chapter.

In addition, as Ostermann and Mezini (2001) point out, the decorator pattern does not account for transparent redirection, that is, the late binding of **this**. Methods overridden in outer decorators thus will never be called by the decorated objects. We say that the decorator performs *forwarding* to its base, not *delegation*.

There are at least two well known solutions to the latter problem (Kniesel, 1999). One solution to achieve late binding is by storing and updating the self-reference in the decorated objects and using the explicitly maintained self-reference instead of **this**. Another possible solution is to modifying the interface of the base class and the decorators, in order to add an explicit *self* parameter to every method.

To the best of our knowledge, there exists no solution in form of an adaptation of the design pattern, that also addresses the former problem of transparency. As we will see in later chapters, the encoding presented in this thesis represents a powerful alternative to the decorator pattern that also takes transparency into consideration.

Pimp-my-Library Pattern. The Pimp-my-library pattern (Odersky, 2006) is a design pattern specific to Scala that allows to *pretend* that methods have been added to a class in retrospect. For this purpose, when a method is not available on an object, *implicit conversions* are used to automatically instantiate a wrapper class that implements the missing interface. Implicit conversions are function calls (or constructor calls as in this case), which are automatically injected by the compiler

at positions that otherwise would have resulted in a static type error. The wrapper class that adds the missing method is very similar to an "implicit decorator". This insight reveals the pimp-my-library pattern as a variant of the decorator pattern, also sharing the same set of disadvantages. Additionally, since the decorator is implicitly generated just for a specific method call, the lifespan of this decorator is usually bound to this single method call – this forbids any stateful computation.

2.2.4 Aggregation Based Language Extensions

Inspired by prototype-based languages Kniesel (1999), Büchi and Weck (2000) and Bettini and Bono (2008) propose extending classical class-based programming languages with a mechanism to automatically delegate calls to the parent / wrapped object. Another language extension proposal by Bettini et al. (2003) adds the decorator pattern as language feature, similar to (Büchi and Weck, 2000), which is then being compiled to standard Java. However, the decorators by Bettini et al. do not meet the transparency requirement and are thus not discussed in further detail.

Darwin / Lava. Faced with third party components or existing legacy software that may not be modified, the goal of *runtime component adaptation* is to enable interaction between those legacy components and components implementing new features. Set in the context of component based software development, Günter Kniesel and his colleagues developed the language Darwin (Kniesel, 1999, 2000) and its Java-like dialect Lava (Costanza et al., 1999) to introduce a type-safe way for component adaptation. The languages draw from both class-based programming languages as well as prototype-based languages. Class-based features include the usual definition and instantiation of classes, inheritance and subtyping. Inspired by prototype-based languages, Darwin also adds primitive language support for delegation, as known from dynamic languages like Self (Ungar and Smith, 1987). With delegation, at runtime a method lookup that fails in the *child object* will automatically continue in the child's *parent objects*.

To specify references to parent objects which should be used for dynamic method lookup, darwin introduces the notion of *delegation attributes*.

```
public class QueryCache extends Caching {
  mandatory delegatee Query parentQuery;
  public QueryCache (Query q) { parentQuery = q; }
  public Result get () {...parentQuery.get ()...} // also performs caching
}
```

The example shows a Darwin class *QueryCache* that defines the delegation attribute *parentQuery* and additionally implements the method *get* as its only method. It is assumed that *get* is also defined in the class *Query*. If the cache lookup fails, *get* is explicitly invoked on the parent to actually perform the query. Since Darwin implements *delegation*, subsequent calls to *get* within *parentQuery* will be dispatched to the late bound receiver.

```
cachingQuery = new QueryCache (myQuery);
```

On *cachingQuery* not only *get* can be invoked, but also all methods inherited from the superclass *Caching* and all methods that are defined in the *declared type* of its delegatee *Query* are available. *QueryCache* is said to be a *declared child class* of *Query*. This shows that there are two ways of creating subtypes in Darwin, by inheritance and by delegation (Kniesel, 1999).

The *QueryCache* example is chosen to highlight the similarities of component adaptation and dynamic specialization. The specialization of a *Query* "object" with caching facilities corresponds to the adaptation of a *Query* "component".

Darwin supports extension of objects at runtime. The object *myQuery* can be equipped with support for caching in retrospect. The extension is permanent, future method calls on *cachingQuery* do not invalidate the extension. Extensions of objects are reflected in the type, facilitating static checks for method calls. Only the requirements of transparency and integration are not met by Darwin. The type of *cachingQuery* is independent of the actual type of *myQuery*. Even if *myQuery* is statically known to be a subtype of *Query*, such as $GroupingQuery <: Query$, applying the extension behaves opaque in Darwin hiding the actual type. Thus the following code will not type check:

new QueryCache (
  new GroupingQuery (`"name"`)).*getGroupingProperty* // *Type Error!*

Since Darwin and Lava are languages on their own right, both languages require a custom compiler. This hinders a seamless integration into existing software projects and a reuse of existing development environments.

Generic Wrappers. Büchi and Weck (2000) introduce *generic wrappers*: a language extension proposal for Java that allows transparent aggregation of objects at runtime. Wrappers are similar to decorators as introduced above[3]. Since wrappers and decorators are essentially the same, they also share the same set of problems. In particular, applying wrappers is not transparent, *per se*. Let us recall the example of multiple applied decorators that does not type check:

new QueryCache (**new** QueryLogger (*query*)).*logMessages* // *Type Error!*

The type error is raised since the method *logMessages* is only defined for objects of type *QueryLogger*, but not for *QueryCache*. The language extension of generic wrappers allows the same example to type-check by adding wrappers as primitives to the language. The two wrappers would be defined as

class *QueryCache* **wraps** Query {...}
class *QueryLogger* **wraps** Query {...}

and the above example then translates to

new QueryCache < **new** QueryLogger < *query* > () > ().*logMessages*

using the special syntax to pass the *wrappee*, the base object to extend. The translated example now does type-check, and rightfully so! Even if the wrappers have been declared to wrap an object of the static type *Query*, they are *generic* in the dynamic type of the wrappee. Hence, *Query* only represents the upper bound of the type of the expected wrappee.

To summarize, the static type of the wrapper is a subtype of the static type of the wrappee. While this is similar to decorators, the difference is, that the runtime type of a wrapped object will also be a subtype of the runtime type of the wrappee. The wrapper is thus fully transparent in its runtime type. Overall, generic wrappers appear to be a good solution to dynamic specialization. Just like decorators they

[3]For the purpose of this thesis, we will treat wrappers and decorators as pseudonyms. To be precise, wrappers describe the general process of object aggregation while decorators describe the modification of behavior.

can be applied at runtime and the wrapping is permanent. Büchi and Weck (2000) claim that the choice of forwarding vs. delegation is orthogonal to their extension proposal. This characterizes their work as focused on transparency, while delegation is of greater importance in (Kniesel, 1999). However, since generic wrappers are a proposed *extension* to Java integrating generic wrappers into an existing code base would at least require to recompile the entire code base. In case of a binary distribution this is impossible.

Incomplete Objects. (Bettini et al., 2004) introduce a calculus of incomplete objects trying to leverage benefits of both class-based as well as object-based programming paradigms. In the calculus, a *mixin* is a function on classes, that can add new methods to the class as well as specify methods which are not yet complete on their own. These methods can either be *abstract* and require implementation, or they can be *redefining* and thus require an original definition that can be referred to via the keyword **next**. A mixin can be applied to a class to yield a subclass, augmented with the functionality defined in the mixin. The novelty of the calculus is, that mixins also can be instantiated to yield *incomplete objects*.

The calculus has been implemented as an extension of Featherweight Java (FJ), called Incomplete Featherweight Java (Bettini and Bono, 2008). There are a few differences between the calculus and the FJ implementation that the authors claim to be necessary to fit the calculus into the setting of FJ:

- The notion of mixins in the calculus translates to *incomplete classes* in IFJ. Similar to mixins, incomplete classes are allowed to contain abstract as well as redefining methods. Likewise, incomplete classes can be instantiated to yield incomplete objects.
- The type system of the calculus is based on collecting constraints about objects which are checked when the object is composed with another object or methods are called. The constraints are collected on the granularity of methods and thus the calculus exhibits a form of *structural subtyping*. Incomplete Featherweight Java in contrast builds on *nominal subtyping*.
- There is also an important semantic difference between the two languages. In the calculus it is possible to invoke complete methods on incomplete objects. Invoking methods that still require completion will lead to an error issued by the type system. In contrast, IFJ forbids method invocation or field access on incomplete objects altogether strictly limiting the usage of incomplete objects before being completed.

We will use Incomplete Featherweight Java (IFJ) for the examples in this section. For instance we can phrase the *QueryCache* example in terms of incomplete objects.

```
class QueryCache abstracts Query {
  void clearCache () {...}
  redef Result get () {...next.get ()...} // also performs caching
}
```

The incomplete class *QueryCache* redefines the method *get*, which is already defined in *Query*. It also adds the new method *clearCache*.

```
QueryCache c = new QueryCache ();
Query q = createQuery ();
c ← q;
```

	Runtime	Transp.	Late Binding	Perm.	Integr.
gbeta*	✓	✓	✓	✓	✗
Static Mixin Comp.	✗	✓	✓	✓	✓
Open Classes	✗	✓	✓	✓	✓[a]
Extension Methods	✗	✓	✗	✓	✓
Decorator	✓	✗	✗	✓	✓
Pimp-My-Library	✓	✓[b]	✗	✗	✓
Darwin	✓	✗	✓	✓	✗
Generic Wrappers*	✓	✓	✓[c]	✓	✗
Incomplete Objects*	✓	✓	✓	✓	✗

(a) Compiles to Java-byte code, but requires custom compiler.

(b) The extension is only transparent, since it is automatically removed after one call.

(c) The authors claim, that both forwarding and delegation are possible semantics but choose forwarding for their formalization.

* Suitable for dynamic specialization.

Table 2.1: Summary of the related work.

The incomplete class *QueryCache* can be instantiated to an incomplete object c. Afterwards, c can be completed by composing it with an instance of a query object q using the composition operator $\leftarrow$. After completion, all methods on an object can be used. At the same time an completed object cannot be composed with other complete objects.

What sounds like a severe limitation, actually does not impose to many restrictions. An complete object can always be "wrapped" into other incomplete objects to be augmented with extensions. Indeed, IFJ is very similar to generic wrappers. Both approaches create new objects that delegate to dedicated parent objects in case a method is not implemented in the extension. Incomplete classes correspond to generic wrappers. Both can be instantiated and applied to a base object (the parent), as becomes visible in the following example:

```
new Buffered < new Stream () > (); // Generic Wrappers
new Buffered () ← new Stream (); // Incomplete Objects
```

In addition to classical subtyping via inheritance, both approaches also add subtyping by delegation to the extended base object. In IFJ the base can be referenced within implementations via the keyword **next**. Generic wrappers offer the semantically equivalent keyword *wrappee*.

However, incomplete objects are first-class citizens within IFJ. They can be passed as function arguments and they can be stored in records, before being applied to augment complete objects. In contrast, the parameter of generic wrappers has to be known at instantiation time of the wrapper.

After all, the calculus of incomplete objects is a transparent solution to dynamic specialization. It supports late binding, extensions are permanent and a static typing discipline ensures that composition of objects and method lookup will not fail.

2.2.5 Summary

The variety of the above listed solutions demonstrates that dynamic specialization (Ernst, 1999) has found much attention in research over the past decades. Being faced

with the limitation of static inheritance, aggregation and delegation offer configuration and adaptation at runtime (Johnson and Foote, 1988). Wrappers and the decorator pattern (Gamma et al., 1994) are early approaches to regain some of the flexibility that has been available for quite some time in dynamic, prototype based languages (Ungar and Smith, 1987). Prototype based languages and the decorator pattern have influenced the design of a few proposed languages extensions ever since (Kniesel, 1999; Büchi and Weck, 2000; Bettini et al., 2003). The notion of incomplete objects (Bettini et al., 2004; Bettini and Bono, 2008) offers similar expressiveness as dynamic specialization. The related work is summarized in Table 2.1.

While three solutions are considered suitable to perform dynamic specialization (marked with $*$), each of them either requires a new compiler or a custom preprocessor. The other solutions do not sufficiently satisfy our requirements.

An encoding enables the usage of a new language feature inside of a host language, without adaption of that host language. This also encourages the reuse of existing libraries and development environments. We also believe that encoding a language feature can aid a deeper understanding of that feature. To fill the gap and enable dynamic specialization in existing languages, this thesis proposes an encoding of extensible functional objects building on a standard encoding of objects as coalgebras.

Chapter 3

Solution

Our solution, in form of a coalgebraic encoding of objects in Scala, is presented in this chapter. Before the solution can be elaborated, Section 3.1 gives some introduction into the terminology of object encodings.

Encoding objects in Scala is like starting from scratch. Users of the encoding cannot use Scala's existing mechanisms to modularly define objects together with our encoding. Thus, Section 3.2 and Section 3.3 build on the encoding of objects as terminal coalgebras and restore some of Scala's expressivity – Section 3.2 introduces composition of coalgebras to allow a primitive form of *statically* modularized object definitions. Section 3.3 then allows to express dependencies between different coalgebraic implementations which corresponds to the feature of self-type annotations in Scala. Section 3.4 adds support for *dynamic specialization*, defined in terms of a modification of *unfold* and composition of coalgebras.

3.1 Background: Encoding Objects

Scala's inheritance and the corresponding typing discipline cannot immediately be influenced by user code. Aside from compiler plugins, which are not discussed in this thesis, it is neither possible to change the semantics of the built-in language constructs nor to change the typing rules which are used to typecheck user programs.

However, encodings allow emulation of language features by describing a precise programming pattern to which the user must adhere. At the same time encodings support a better understanding of features by expressing their semantics in an already studied host-language.

One famous example is the Church encoding. By interpreting data as program that is parametrized over the constructors of the data type, it is possible to encode data types within the λ-calculus. For instance the number 3 is represented by the Church numeral:

$$three = \lambda z.\lambda s.s\,(s\,(s\ z))$$

For instance, invoking the Church encoded value *three* with `""` and $\lambda s.s + \texttt{"I"}$ yields the unary representation `"III"`. Using the Church encoding, the feature "data-types" can be used in any host-language that only supports first-class functions. (For more information about Church encodings we refer to (Pierce, 2002)).

This example of en encoding showed, that we can implement new language features or re-implement and modify existing ones by encoding them as user programs of the host language. This style of language extension has the advantage that it can be distributed as a single (possible empty) library coupled with a programming pattern. At the same time the semantics of existing programs written in the host language is not affected.

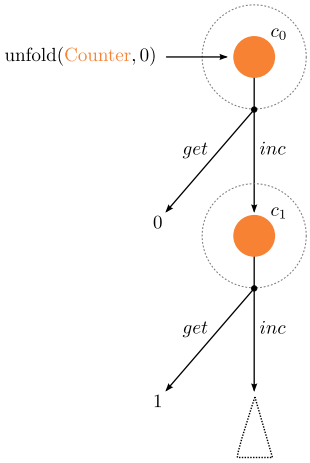
Object oriented programming has been encoded in functional programming languages (Cardelli, 1984), and functional programming has been encoded in object oriented languages (Abadi and Cardelli, 1996, p.66). Oftentimes, the programming

```
trait Counter {
  def get: Int
  def inc: Unit
}
```

(a) The interface of a counter object.

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

(b) The interface endofunctor representing the *Counter* interface.



(c) An infinite tree of observations on a counter object.

Figure 3.1: The interface of an object, its endofunctor and the corresponding infinite tree of observations.

paradigm that is not natively supported, is emulated as an encoding. The encoding presented in this thesis is different in this regard. Scala already offers good support for both programming paradigms, that is, functional programming and object oriented programming. However, since we cannot change the built-in semantics of objects in Scala to also support the one missing feature of dynamic specialization, we make use of the flexibility of encodings and design the object system from scratch. In consequence, existing modularity features in Scala like self-type annotations or mixin composition cannot immediately be reused in our encoding. Instead, we have to reimplement these features in our encoding.

In the remainder of this section we will review a standard encoding of purely functional objects as coalgebras as well as the necessary preliminaries. This encoding will serve not only as theoretical background but also as starting point for further developments in later sections.

3.1.1 Objects as functions

Objects can be described as a collection of operations, bundled with private state, which is individual to each object. The only way to interface with the objects is via the collection of operations and only their implementation is allowed to access the object's private state.

The first purely functional encoding of objects was introduced by Cardelli (1984). As noted above, a central concept of object oriented programming is that of private state. Details of the implementation might be hidden to protect the user from future changes as well as facilitating correctness proofs. According to private state we can distinguish at least two different kinds of functional object encodings, which are sketched in the remainder of this section. As an introductory example we use a counter object that provides two methods: *get* to retrieve the current count and *inc* to increment the count by one. The Scala trait defining the interface of such a counter is depicted in Figure 3.1a.

The first encoding interprets objects as records of functions that close over the private (mutable) state (Cardelli, 1984). This encoding is for instance predominant

in JavaScript to simulate private state and perform information hiding. A simple counter in this style might be implemented as

```
type Counter = (Unit ⇒ Int, Unit ⇒ Unit)
def makeCounter (n: Int) = {
  var count = n
  ((() ⇒ count, () ⇒ count += 1)
}
```

Building a fresh counter with initial state set to zero and increasing it two times amounts to

```
val c = makeCounter (0); c._2 (); c._2 ()
```

The second kind of encoding does not use mutable references to express the changed state of an object. Instead, a modified copy of the state is returned (Pierce, 2002). In this scenario the state needs to be threaded through all the method invocations. To allow information hiding, the type of the state is represented by an existential.

```
type Counter = (S, (S ⇒ Int, S ⇒ S)) forSome {type S}
```

The existential type is necessary to pass on the modified state without knowing its actual type. However, we will not go into the details of this encoding, but instead we will see a variant of this encoding based on coalgebras.

3.1.2 Objects as coalgebras

Before we can get to the actual encoding, some preliminaries on coalgebras have to be discussed. Regardless of their theoretic foundation in category theory, coalgebras are a strikingly simple concept. Informally, just like an object, a coalgebra can be thought of as a black-box. By the principle of information hiding, we do not know about the internal workings of the black box, but only can observe its behavior from the outside. It hence describes a state-based system, where the state-space is hidden from the external observer (Jacobs and Rutten, 1997).

Returning to the counter example above, we notice that we only can analyze the current state of the counter object by invoking the *get* method or trigger a change of state (a state transition in terms of state machines) by invoking the *inc* method. Performing these actions repeatedly leads to the infinite tree of observations depicted in Figure 3.1c. The object in state c_1 is retrieved by calling *inc* once after unfolding. The subtree rooted at observation $c_1.inc$ represents the infinite tree of observations that will always have the same shape.

Thus, an object itself might as well be defined by its observable behavior, or as we will see by its *terminal coalgebra* (Reichel, 1995).

For our purposes, we define a coalgebra as a function $S \Rightarrow F[S]$, as defined in Figure 3.2a. Here F is a type constructor with kind $* \to *$ and S is the type of the state. A coalgebra can thus be seen as a one step function from a state to the structure of next possible states.

Taking the interpretation of a type as the set of its inhabitants, in the categorical setting F represents an endofunctor (a functor whose domain and co-domain are the same) on the category **Set** while S is called the *carrier set* of the coalgebra. To capture the notion of functors more precisely we introduce the type class[1] *Functor*

[1]Type classes are a language feature heavily used in the programming language Haskell. We use an encoding of type classes in Scala introduced by Oliveira et al. (2010). For this purpose, a type class $T[\_]$ is represented by a trait T, while an instance of $T[A]$, for a specific A, is interpreted as evidence that A is a member of the type class T.

in Figure 3.2b. For a given structure described by F, members of the type class *Functor* allow lifting an operation $A \Rightarrow B$ to an operation between the structures $F[A] \Rightarrow F[B]$. Informally, a functor instance allows us to modify elements inside a structure, without having to know the structure itself.

For convenience we also define the function *map*, that uses implicit resolution to find evidence for the corresponding functor:

$$\textbf{def}\ \mathit{map}[F[\_]\colon \mathit{Functor},\ A,\ B]\,(\mathit{fa}\colon F[A])\,(f\colon A \Rightarrow B)\colon F[B] = \\ \quad \text{implicitly}[\mathit{Functor}[F]].\mathit{map}\,(\mathit{fa})\,(f)$$

The syntax $F\,[\_]\colon \mathit{Functor}$ is called a *context bound* and is rewritten by the Scala compiler to an additional argument prepended to the *implicit argument list*, which itself is appended to the argument lists if not already existent. The result of the rewriting in this case would be:

$$\textbf{def}\ \mathit{map}[F[\_],\ A,\ B]\,(\mathit{fa}\colon F[A])\,(f\colon A \Rightarrow B)\,(\textbf{implicit}\ \mathit{ev}\colon \mathit{Functor}[F])\colon F[B]$$

The context bound syntax will be used throughout this thesis, since it provides us with a concise notation for expressing required type classes.

Following (Jacobs, 1995; Lämmel and Rypacek, 2008) we can express the interface of *Counter* from Figure 3.1a by the *interface endofunctor* $\mathit{Counter}_F$:

$$\begin{array}{l}\textbf{trait}\ \mathit{Counter}_F[S]\ \{\\ \quad \textbf{def}\ \mathit{get}\colon(\text{Int},\ S)\\ \quad \textbf{def}\ \mathit{inc}\colon(\text{Unit},\ S)\\ \}\end{array}$$

In both methods *get* and *inc* the return type is tupled with the type parameter S indicating the modified state. Without loss of generality and for ease of presentation we further simplify the interface functor by distinguishing between methods which are *observations* and those that are *transformations*. In general, also methods that are observations can trigger a change of the internal state, but we assume that this is not the case. However, the full expressiveness can easily be restored by splitting every method into a transformational and a observational counterpart and agreeing on the calling convention to always call the two in sequence. The simplified interface is shown in Figure 3.1b.

An implementation of the *Counter* interface now can be given as an instance of the coalgebra with the functor set to $\mathit{Counter}_F$ and the type of the state set to Int.

$$\begin{array}{l}\textbf{val}\ \mathit{Counter}\colon \mathit{CoAlg}[\mathit{Counter}_F,\ \text{Int}] = s \Rightarrow \textbf{new}\ \mathit{Counter}_F[\text{Int}]\ \{\\ \quad \textbf{def}\ \mathit{get}\colon \text{Int} = s\\ \quad \textbf{def}\ \mathit{inc}\colon \text{Int} = s + 1\\ \}\end{array}$$

The result-type of *get* and *inc* only match by coincidence since the type parameter S is instantiated with Int. It becomes visible, that the coalgebra *Counter* describes just a single computational step. More precisely, it describes how to create observational structure $\mathit{Counter}_F$ from a given state S. To instantiate a new counter and increment it by two we provide the initial state 0 and repeatedly apply the coalgebra *Counter*:

$$\mathit{Counter}\,(\mathit{Counter}\,(\mathit{Counter}\,(0).\mathit{inc}).\mathit{inc}).\mathit{get}$$

Even in this simple example we can see two things that might be undesirable. Firstly, the *unfolding* of the observational tree is left to the user of our counter object by repeatedly applying the *Counter* coalgebra. Secondly, the type of the internal state

```
type CoAlg[F[_], S] = S ⇒ F[S]
```

(a) A coalgebra is a function from a given state to a structure of state.

```
trait Functor[F[_]] {
  def map[A, B]:(A ⇒ B) ⇒ (F[A] ⇒ F[B])
}
```

(b) The function *map* allows lifting an operation $A \Rightarrow B$ to an operation between functors $F[A] \Rightarrow F[B]$.

```
trait Fix[F[_]] {
  def out: F[Fix[F]]
}
```

(c) The greatest fixed point *Fix* represents the terminal F-coalgebra. The infinite unfolding is hidden under the lambda abstraction of *out*.

```
def unfold[F[_]: Functor, S](co: CoAlg[F, S]): S ⇒ Fix[F] = state ⇒ new Fix[F] {
  lazy val out = map(co(state))(unfold(co))
}
```

(d) Unfolding a coalgebra with an initial state results in the greatest fixed point. This corresponds to object initialization.

Figure 3.2: Encoding objects as coalgebras.

Int is exposed as part of the type of the coalgebra $CoAlg[Counter_F, \text{Int}]$. Neither does this comply with information hiding, nor does it match our description of objects from above.

To overcome these issues we introduce the notion of terminal coalgebras. For our purposes, we define the *terminal coalgebra* to be the greatest fixed point of the function $S \Rightarrow F[S]$. It thus represents the unfolded, infinite tree of observations for the recursively expanded functor F. Informally, the terminal coalgebra for a functor F can thus be derived by infinite substitution of occurrences of the carrier within the structure by the next layer of unfolding. The definition of the fixed point $Fix[F]$ for the functor F in Scala is given in Figure 3.2c. It is important that *out* is a method (and not an eagerly computed value) in order to assure that the infinite expansion is only performed lazily one layer after another.

In the coalgebraic encoding, objects are thus represented by the greatest fixed point of the endofunctor interface F. For the *Counter* example above we can define the convenience type alias

```
type Counter = Fix[Counter_F]
```

Figure 3.2d defines *unfold* as a Scala function[2]. Given any coalgebraic implementation $CoAlg[F, S]$ we can instantiate objects by *unfolding* the coalgebra to its infinite tree of observations. As mentioned above, the unfolding is performed lazily for every single layer, triggered by a call to *out*. The recursion scheme used for unfolding is also known as *anamorphism* (Meijer et al., 1991). After unfolding, the initial state S is not visible anymore and thus the fixed point serves a similar purpose as the existential type in the sketched functional encoding above.

Finally, we can define a function that given the initial state allows us to build objects of type *Counter*.

```
def makeCounter(n: Int) = unfold[Counter_F, Int](Counter, n)
```

A counter object that is constructed this way can be incremented two times as follows:

```
makeCounter(0).out.inc.out.inc.out.get
```

[2]For ease of presentation we will use both the curried variant *unfold* (*co*) (*s*) and the uncurried variant *unfold* (*co*, *s*).

```
def compose[F[_]: Functor, G[_]: Functor, S1, S2] (
  coF: CoAlg[F, S1], coG: CoAlg[G, S2]) = mix (
    map (coF (s ._ 1)) ((_, s ._ 2)),
    map (coG (s ._ 2)) ((s ._ 1, _))
  )
```

Figure 3.3: Basic form of coalgebra composition. Leveraging the fact that F and G are functors we can tuple the state after applying the coalgebras point-wise. The result is a coalgebra implementing both interfaces.

In the remainder of this thesis, often the call to *out* will be omitted for conciseness of presentation. In fact, using implicit conversion in Scala this is also often the case for the actual encoding.

To summarize, objects can be encoded coalgebraically by translating their interface to an endofunctor F. Implementations of these interfaces then correspond to F-coalgebras. Given an initial state S by infinite unfolding of any coalgebra $CoAlg[F, S]$ we can compute the terminal coalgebra as the greatest fixed point $Fix[F]$. The next layer of unfolding can be unrolled by a call to *out*. This allows making observations using the methods specified by the interface functor F.

3.2 Composition of Coalgebras

In Section 3.1 we have seen how objects can be encoded using coalgebras to describe their behavior. This and the following sections will stepwise elaborate on the encoding to finally arrive at modular and extensible description of objects, embedded into Scala.

In Scala it is possible to define traits A and B in isolation and then statically compose them to get the intersection type A **with** B. At the current stage of our encoding this is not supported anymore. We can define two coalgebras describing the implementation for A and B, respectively. However, at this stage of development we cannot compose these two coalgebras to an implementation for A **with** B. The current section augments the standard coalgebraic encoding of objects to restore Scala's expressiveness with regard to static mixin composition.

While it is possible to combine the two interface functors A_F and B_F via

$$\textbf{type } Both_F[X] = A_F[X] \textbf{ with } B_F[X]$$

we cannot just compose two coalgebras describing the implementation of the two interfaces.

Recalling that a coalgebra is defined as a *function* from the current state to the implementation of the next possible observations helps understanding why this is difficult. Two coalgebras, *coA* and *coB*, defined over two *different* functors (A_F and B_F) and two *different* types of state (S_1 and S_2) are thus given by the functions

$$\textbf{val } coA: S_1 \Rightarrow A_F[S_1] = ...$$
$$\textbf{val } coB: S_2 \Rightarrow B_F[S_2] = ...$$

But how can we combine two the coalgebras *coA* and *coB* then? At first, we will consider the case where *coA* and *coB* are semantically unrelated and thus do not depend on each other. We will later see how the encoding can be augmented to allow expressing dependencies between coalgebraic specifications. Our goal is to derive a coalgebra *coBoth* that, given both states S_1 and S_2, implements observations for the intersection $Both_F$:

val *coBoth*:$(S_1, S_2) \Rightarrow Both_F[\![S_1, S_2)]$ = ...

Following the types, we can see both states are provided as a tuple, so we can apply the coalgebras point-wise to their corresponding state resulting in values $A_F[S_1]$ and $B_F[S_2]$. However, the question remains how the result of the point-wise applications can be combined to yield the intersection type $Both_F$. To answer this question, we will assume the existence of the Scala function *mix* with the signature

def *mix*$[A, B]\,(a\colon A, b\colon B)\colon A$ **with** B

that allows composing two arbitrary Scala values to their intersection type.

Using *mix* the implementation of *coBoth* now seems straightforward:

val *coBoth* = $(s\colon(S_1, S_2)) \Rightarrow$ *mix* (*coA* $(s$._1$)$, *coB* $(s$._2$)$)

However, the type of *coBoth* above is $(S_1, S_2) \Rightarrow A_F[S_1]$ **with** $B_F[S_2]$. This is not the result we were trying to achieve. The resulting structure of each component is only defined over the corresponding state, not the tuple representing the common state. Requiring A_F and B_F to be functors, we can tuple the result with the original residual state. This is safe, since the state is *private* to each individual coalgebra. Since we required the two coalgebras to be semantically unrelated, state changing operations in one coalgebra do not affect the state within the other. Using the knowledge that A_F and B_F are functors we can modify the above definition to

```
implicit val funcA: Functor[A_F] = ...
implicit val funcB: Functor[B_F] = ...
val coBoth = (s:(S_1, S_2)) ⇒ mix (
  map (coA (s._1)) ((_, s._2)),
  map (coB (s._2)) ((s._1, _))
)
```

The functor instances *funcA* and *funcB* are marked as implicit parameters to allow omitting them for the calls to *map*.

Generalizing this example, the point-wise composition of two coalgebras now can be expressed as in Figure 3.3.

Finally, using *compose*, the Scala code of statically composing two traits A and B and instantiating the result

new A **with** B $\{/*$ ... *s1* ... *s2* ... $*/\}$

can be translated to our encoding as follows:

unfold (*compose* $(A, B), (s1, s2)$)

While in the Scala code A and B are traits and thus second class language features, in our encoding A and B are coalgebras and hence first-class values that can be passed as arguments to *compose*.

Now that we have seen how coalgebras can be composed, the remainder of this section will discuss some technical details of the implementation.

3.2.1 Automatic Functor Materialization

We have seen that functors play an important role in our encoding. However, manually writing functor instances for every single interface in the user program can be tiresome and imposes additional weight on the usage of the encoding. The implementation of

```
trait With[A, B] {
  type Apply = A with B
  def apply (a: A, b: B): Apply
}
```

(a) Instances of the type class A **With** B witness that values of type A and B can be composed.

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

(b) Instances of the type class F $\mathbf{With_F}$ G witness that *for every* A values of type $F[A]$ and $G[A]$ can be composed.

```
def mix[A, B] (a: A, b: B) (implicit ev: A With B): A with B = ev (a, b)
```

(c) The function *mix* is implemented by implicit search for in instance of **With**

Figure 3.4: The type classes **With** and $\mathbf{With_F}$ that can be used to witness object composition.

a functor instance on the one hand is probably semantically unrelated to the user program and on the other hand is quite mechanic to execute. Especially the latter is the reason why the code for this thesis comes with a macro implementation that allows automatic derivation of functors. Using implicit macros (Burmako, 2013), that is, macro calls that are automatically inserted by the compiler, functor instances can be implicitly materialized and passed as arguments to *compose*. This allows the following lightweight syntax of calling the *compose* method

$$compose\,(coA,\, coB)$$

which is expanded by the compiler to:

$$compose[A_F,\, B_F,\, S_1,\, S_1]\,(coA,\, coB)\,(functor[A_F],\, functor[B_F])$$

Here the method call to *functor* in turn invokes the macro implementation, using reflection on the types of A_F and B_F to generate instances of $Functor\,[A_F]$ and $Functor[B_F]$, respectively.

3.2.2 Composition of Scala Values

In the beginning of this section we have used the method *mix* to compose two values a and b (of type A and B, respectively) to the intersection A **with** B. In this subsection we will see how the function *mix* actually can be implemented.

In their work on feature oriented programming with object algebras Oliveira et al. (2013) introduced a reflective composition operator *delegate*[3] that like *mix* in this thesis allows to compose two objects to their intersection. Their implementation is based on a dynamic proxy, selecting the object to forward a method call to at runtime. The instantiation of the dynamic proxy object however comes with the restriction that a nominal subtype of the intersection type has to be provided as additional type argument $S <: A$ **with** B. This is necessary in order to use the function *createInstance* from the Java reflection API, that does not offer native support for intersection types (Oliveira et al., 2013).

Faced with the same problem of composing two objects by forwarding, in earlier work (Rendel et al., 2014) we introduced the type class **With** as defined in Figure 3.4a. An instance of $\mathbf{With}\,[A,\, B]$[4] is used as evidence that values of type A and B can

[3] The function is called delegate, but actually implements *forwarding*.

[4] **With** is highlighted different than the other types since we will mostly use it infix A **With** B (which is legal syntax in Scala), reminiscent of the Scala intersection type A **with** B it computes.

```
trait A {
  def foo: Int
}
```

(a) Example trait A.

```
trait B {
  def bar (n: Int): String
}
```

(b) Example trait B.

```
implicit object ab extends (A With B) {
  def apply (a: A, b: B) = new (A with B) {
    def foo = a.foo
    def bar (n: Int) = b.bar (n)
  }
}
```

(c) Composing instances of A and B by forwarding.

Figure 3.5: Creating the evidence A **With** B by hand.

be composed. The witness is provided by the method *apply* that, given the two objects a and b, actually computes the composed result. In order to avoid having to manually implement A **With** B for every A and B that need to be composed, implicit macros are used to automatically generate type class instances, similar to the functor materialization described in Section 3.2.1.

Macros and the type class machinery help avoiding the problem of having to use the Java reflection API and thus render specifying a nominal subtype unnecessary. The implementation in (Rendel et al., 2014) however comes with the technical limitation that the traits A and B must only contain value declarations or method declarations without any arguments. This was fine for their work on object algebras since the interfaces that had to be combined were always very simple. For combining interface endofunctors of arbitrary objects however, this imposes a serious limitation. The implementation of mixin composition complementing this thesis builds on the method *mix* presented in (Rendel et al., 2014) but removes the above mentioned limitation.

Let us now see how the method *mix* is implemented in this thesis. Figure 3.4c reveals that in addition to the two objects that should be mixed together, *mix* also takes an implicit evidence that values of A and B can be composed. The advantage of this design is that there are multiple ways of providing such evidence.

Manual evidence The first way to provide evidence is creating a value of type A **With** B by hand. This of course should only be considered in individual cases where a specialized composition is required. Inspecting an example of manual evidence however also serves as a good introduction on how the composition operation works. Figures 3.5a and 3.5b define two simple example traits A and B, both consisting only of one member each. The instance of A **With** B that serves as evidence that A and B can composed, is given in Figure 3.5c. On a call to *apply* a new instance of the intersection type A **with** B is created by forwarding the method implementations to the provided instances a and b, respectively. In case that a method m is define in both A as well as B the one provided last in left-to-right order (in this case B) "wins". The composition operation thus is right-biased which will be important later, when considering method overriding.

Macro evidence Building on the macro introduced in (Rendel et al., 2014) a more convenient way of providing evidence A **With** B is by generating the instance at compile-time. A necessary condition to be able to do so, is that both types A and B are fully known at the compile time of the call to $mix\,[A, B]$. If this is the case, the macro can analyze both types A and B via compile time reflection and thus automatically generate the instance that is given by hand in Figure 3.5c. This is for example not the case if one of the two is a type parameter and thus only known when instantiated. The version of the macro prepared for this thesis now also supports

methods with arguments (even multiple argument-lists) such as the method *bar* in Figure 3.5b.

Reflective evidence As an alternative to the macro based evidence, the instance of A **With** B can also be generated at runtime using Scala's mirror based reflection API. With the macro based evidence generation, both types A and B have to be statically known at compile-time when calling $mix[A, B]$. With reflective evidence in contrast the type information can be provided dynamically by providing *type-tags* for A and B with the call to *mix*. Type-tags are a compiler generated physical manifestation of types, available at runtime and thus outlive the compiler phase of erasure. Type-tags can be implicitly acquired from the compiler just as type classes by using the context bound syntax:

```
def canCompose[A: TypeTag, B: TypeTag](a: A, b: B, ...) = {
  ...
  mix(a, b)
  ...
}
```

However, the main advantage of reflection over macros in our setting is the individual handling of $TypeTag[A]$ and $TypeTag[B]$. This becomes visible when currying the function *canCompose*:

```
def canCompose[A: TypeTag](a: A) = new {
  def apply[B: TypeTag](b: B, ...) = {
    ...
    mix(a, b)
    ...
  }
}
```

This allows successive acquisition and storing of the type-tags, necessary for a call to *mix*. This is not possible with the macro based implementation. The flexibility to select between the three ways of providing evidence has proven useful over the course of implementing the coalgebraic encoding.

The last important improvements of the composition implementation presented here is the generalization of **With** to type-constructors of kind $* \to *$.

For this purpose, the constructor class $\mathbf{With_F}$ is defined as in Figure 3.4b. An instance of F $\mathbf{With_F}$ G is thus a rank-2 polymorphic evidence, proving that *for all* types A an object of $F[A]$ and an object of $G[A]$ can be composed to $F[A]$ **with** $G[A]$. The implementation of automatically materializing instances of $\mathbf{With_F}$ is technically more involved then spawning instance of **With** but conceptually the same. Instances of $\mathbf{With_F}$ are important to generically compose two different interface endofunctors without knowing their state type, yet. The type member $Apply[A]$ will be used in the following to avoid the need for type-aliases such as $Both_F$ above. Using type member access, $Both_F$ can be expressed inline as: $(A_F\ \mathbf{With_F}\ B_F)_{\#\mathrm{Apply}}$. In the remainder, the term *intersection (type) of functors* will refer to this type application of $(\mathbf{With_F})_{\#\mathrm{Apply}}$[5].

To summarize, compared to Rendel et al. (2014) the implementation developed for this thesis *a*) adds support for methods with arbitrary arguments, *b*) adds an

[5]For ease of presentation, we will sometimes use the more lightweight A_F **with** B_F to mean the intersection of two functors $(A_F\ \mathbf{With_F}\ B_F)_{\#\mathrm{Apply}}$. Technically the lightweight version is incorrect since **with** expects its arguments to be types and not type constructors. The usage of this abbreviation will thus be reduced to in-text usage, only.

```
type OpenCoAlg[Self[X] <: Prov[X], Prov[_], S]
  = CoAlg[Self, S] ⇒ CoAlg[Prov, S]]
```

(a) The type of open coalgebras, allowing to express dependencies over the type of the self-reference.

```
def compose[
  Self_1[X] <: Prov_1[X], Prov_1[_], S_1,
  Self_2[X] <: Prov_2[X], Prov_2[_], S_2] (
  co_1: OpenCoAlg[Self_1, Prov_1, S_1],
  co_2: OpenCoAlg[Self_2, Prov_2, S_2]):
  OpenCoAlg[
    (Self_1 With_F Self_2)_#Apply,
    (Prov_1 With_F Prov_2)_#Apply,
    (S_1, S_2)
  ]
```

(b) An intermediate version of the signature of the function *compose*, enabling composition of open coalgebras.

Figure 3.6: First attempt at defining composition of coalgebras with (mutual) dependencies.

additional reflection based implementation available under a unified API and thus *c*) supports both macro and reflection based implementations which can be used in parallel due to shared type classes **With** and $\mathbf{With_F}$; the latter *d*) allows composition of type constructors using rank-2 types.

3.3 Expressing Dependencies

In the beginning of the last section we have seen how separately defined coalgebras for different functors can be composed to yield a coalgebra for the intersection of the functors. However, we required them to be semantically unrelated and thus the two coalgebras could not refer to each others definitions, hindering reuse of implementation. In class-based object oriented languages it is common to specify dependencies on other components over the type of the late-bound pointer to *self*. Dependencies are either expressed structurally by marking members as **abstract** or nominally by extending other implementations or adding required interfaces to the *self-type* (Odersky and Zenger, 2005). It is important to distinguish the two. While the use of the **extends** keyword explicitly binds the reference **super** to the specified class, using self-type annotations we only express an upper bound of the type of the reference *self*. The requirement can be fulfilled by a superclass, by a subclass or even by the class itself that is being defined. In this section we will focus on dependencies as they are expressed by self-types annotations.

The current definition of coalgebras only consists of the type of the state and the interface endofunctor. Thus it is not possible to express dependencies to other coalgebras in the type of a coalgebra as it is defined at the moment. Like we have seen above, dependencies can be articulated over the self-type, that is, requirements on the type of *self*. The special variable *self* is bound late by constructing the fixed point of a function $Self \Rightarrow Self$ at the time of object creation. We can use this idea of linking components over the self-reference to allow expressing dependencies between coalgebraic components.

Modeling open-recursion via a fixed point construction is a well known technique (Cook and Palsberg, 1989; Pierce, 2002; Oliveira et al., 2013). In the context of object algebras, Oliveira et al. (2013) introduced a type *Open* that allowed imposing requirements on the self-reference of an object algebra.

```
trait Skip_F[S] {
  def skip: S
}
```

(a) Interface functor.

```
val SkipC: OpenCoAlg[(Counter_F With_F Skip_F)_#Apply, Skip_F, Unit]
  = self ⇒ state ⇒ new Skip_F[Unit] {
    def skip = self (self (state).inc).inc
  }
```

(b) Coalgebraic implementation of the skipping counter.

Figure 3.7: Interface functor and an implementation as *OpenCoAlg* for a *Counter* with "skipping" functionality.

3.3.1 Dependend Coalgebras: a First Attempt

Following (Oliveira et al., 2013), we can introduce the type *OpenCoAlg* as in Figure 3.6a. The upper bound $Self <: Prov$ guarantees that all methods available on the provided interface can also be used on the self-reference. That is, given the coalgebra $CoAlg\,[Self, S]$ representing the overall aggregate of all component coalgebras and the current state of type S we can compute the provided interface $Prov\,[S]$. Since the self-reference is bound late, it can contain implementations from other coalgebras that have been composed with the current one. Hence, single instances of *OpenCoAlg* can in general be incomplete, reminiscent of incomplete classes in (Bettini and Bono, 2008).

This definition looks fine at the first glance so we can try to define an updated version of the function *compose* for two instances *OpenCoAlg*. The signature of *compose* can be seen in Figure 3.6b.

Both, the type of the self-references, as well as the type of the provided interface are *aggregated* by means of the intersection operator $\mathbf{With_F}$. By aggregating both sides, the provided interfaces of the coalgebras can mutually implement the required interfaces. Loosely speaking, both required as well as provided interfaces will eventually level up to the same set of interfaces, allowing the self reference to be closed. Since for creating the fixed point a function $Self \Rightarrow Self$ is required, we only can close the self-reference when *Self* and *Prov* agree.

When trying to implement this new version *compose* we could chose the same strategy as in Figure 3.3: Projecting into the state before applying a coalgebra and then mapping the result back to the tupled state. This implementation strategy however has some problems as becomes visible in the following simple example.

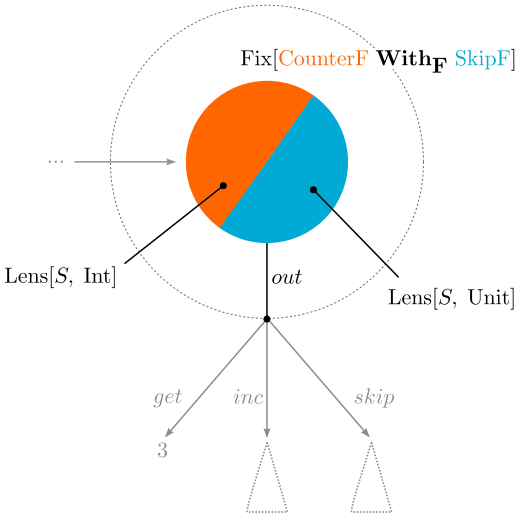
Let us extend the *Counter* example with an additional interface $Skip_F$ (defined in Figure 3.7a), enhancing the counter with a method *skip*, that should invoke *inc* twice to skip one intermediate count. We can implement $Skip_F$ coalgebraically by means of *OpenCoAlg*, as in Figure 3.7b. Composing *SkipC* with a $Counter_F$ coalgebra immediately reveals a problem: A call to *skip* only increases the counter once.

Why is this the case? The type of *state* only accounts for one slice of the state relevant for the single component at hand, here Unit. Changes to other parts of the object state are simply ignored by the assumed implementation of compose that uses tupling as in Figure 3.3[6].

3.3.2 Taking the Residual State into Account

The previous attempt of defining *compose* showed that it is not enough to consider only the part of the state that is in focus when defining a particular open coalgebra. The state of an object that has been instantiated from the result of composing multiple

[6]The implementation strategy of tupling the state would work for the subset of coalgebras that use *self* at most once in each method implementation.



(a) Using lenses to access private slices of state of a "skip-counter-object".

trait *Lens*[S, T] {
  def *get* (s: S): T
  def *set* (s: S, t: T): S
}

(b) Signature of a lens to focus T within S.

Figure 3.8: Lenses can be used to focus on a private slices of state in the general frame, the unknown residual.

coalgebras consists of many disjoint slices that are private to the individual coalgebras. Since method invocations performed via the self-reference might lead to changes of different parts of the state, it is not enough to just take the private state of each individual coalgebra into account. We also have to consider the *frame*[7] representing the rest of the object state, in which the change to the state occurs (private to some other module)

However, when defining an open coalgebra it is impossible to know what the frame will be, since the coalgebra could be composed with another one, defined in a separate module. Hence, the definition has to be valid for *all* possible frames.

To this end, we introduce the notion of a *FramedDefinition*.

trait *FramedDefinition*[*Def*[_], *State*] {
  def *apply*[*Frame*] (*priv*: *Lens*[*Frame*, *State*]): *Def*[*Frame*]
}

A framed definition allows specification of the private state, as part of a context (unknown by parametricity) of type *Frame*. The "as part of" relation is modeled using a *Lens*. Lenses are a programming concept heavily used in the programming language Haskell. Figure 3.8b defines a *Lens*[S, T] as a pair of methods *get* and *set*[8]. In our context, the first method, *get* allows projecting the private slice of type T out of a general frame of type S, while the second one allows updating the projected part inside a frame with a new slice of type T. In addition we assume that composition of lenses is defined, such that a lens *l1* with type *Lens*[S, T] and a lens *l2* with type *Lens*[T, U] can be composed via *l2 compose l1* to a lens of type *Lens*[S, U].

[7] We have chosen the name *frame* to match a similar concept in the field of artificial intelligence, as pointed out by Tillmann Rendel in personal communication.

[8] The interface *Lens* is reminiscent of a coalgebra for a constant type T. And rightfully so: Currying the state S and fixing T we get the coalgebra $S \Rightarrow$ *LensF* [S] with **trait** *LensF* [S] {**def** *get*: T; **def** *set* (t: T): S}. The exact relationship between lenses and coalgebras is described by Gibbons and Johnson (2012) and will not be of further interest, here.

A framed definition can thus be interpreted as:

> *Given a lens that allows accessing a known part of the state within a frame, a definition in the general frame can be provided.*

Equipped with framed definitions, open coalgebras can be defined within a frame (representing the overall object state) as follows:

```
type OpenCoAlg[Self[X] <: Prov[X], Prov[_], S] =
  FramedDefinition[({
    type Apply[T] = CoAlg[Self, T] ⇒ CoAlg[Prov, T]
  })#Apply, S]
```

Inlining *FramedDefinition*, the same type can equivalently be given by the slightly more readable variant, defined in Figure 3.9a. Equipped with this new definition of *OpenCoAlg* we now can implement the compose operation as in Figure 3.9b. The implementation assumes the presence of the two lenses *first* and *second*, corresponding to the first and second projection into a tuple. The result of the composition again is a open coalgebra with aggregated self-types and provided interfaces.

Figure 3.8a visualizes an object that has been instantiated from two coalgebras, one contributing the implementation for $Counter_F$ and the other contributing the implementation for $Skip_F$. The state S thus of two private slices that can be accessed by the coalgebras via $Lens[S, \text{Int}]$ and $Lens[S, \text{Unit}]$, respectively.

3.3.3 Closing Open Coalgebras

In order to define *unfold* for open coalgebras (to ultimately allow instantiating objects) we need a means to close the self-references. To this end, following (Oliveira et al., 2013), we can use a lazily computed fixed point expressed in terms of *fix* which is defined as follows:

```
def fix[R](f:(⇒ R) ⇒ R): R = {lazy val res: R = f(res); res}
```

Here, the argument type of f is marked as being lazily evaluated (by prefixing the type with ⇒), which is necessary to prevent non-termination.

Only those open coalgebras where *Self* and *Prov* are instantiated with the same interface functor F can be closed. The reason is that only those coalgebras can be passed as self-reference to themselves, self-sufficiently satisfying the required interface with the interface they provide. We call this subset of open coalgebras *complete coalgebras* – a type alias *CompleteCoAlg* is defined in Figure 3.9c.

Figure 3.9c defines the function *close* that takes a $CompleteCoAlg\ [F, S]$ and returns a $CoAlg[F, S]$ by closing the self-reference. The closing is achieved by computing the fixed point of the complete coalgebra[9]. The function *id* is defined as the identity lens with type $Lens[R, R]$ for all R.

Finally, we can give the definition for the function *unfold* in Figure 3.9c. Compared to the definition of *unfold* in Section 3.1, the only change that has to be made is closing the complete coalgebra before applying the state to it.

Using the new definition of open coalgebras we now can redefine the skip counter implementation *SkipC* as in Figure 3.9d and compose it with the correspondingly redefined *Counter* to obtain a complete coalgebra. The complete coalgebra in turn can be instantiated using the initial state $(0, ())$. The last line of the following example,

[9]The parameters *self* and *state* are repeated, as opposed to the shorter version *fix* $\{ co[S]\,(id) \}$, in order to allow the compiler to infer a lazy argument type for *self*

```
trait OpenCoAlg[Self[X] <: Prov[X], Prov[_], S] {
  def apply[T](priv: Lens[T, S]): CoAlg[Self, T] ⇒ CoAlg[Prov, T]
}
```

(a) Open self-references allow specifying dependencies on the self-type.

```
def compose[
  Self_1[X] <: Prov_1[X], Prov_1[_], S_1,
  Self_2[X] <: Prov_2[X], Prov_2[_], S_2](
  co_1: OpenCoAlg[Self_1, Prov_1, S_1],
  co_2: OpenCoAlg[Self_2, Prov_2, S_2])
= new OpenCoAlg[(Self_1 With_F Self_2)_#Apply, (Prov_1 With_F Prov_2)_#Apply, (S_1, S_2)] {
  def apply[T](priv: Lens[T, (S_1, S_2)]) = self ⇒ state ⇒ {
    val left = co_1[T](first compose priv)(self)
    val right = co_2[T](second compose priv)(self)
    mix[Prov_1[T], Prov_2[T]](left(state), right(state))
  }
```

(b) Composing two separately defined coalgebras with (possibly) mutual dependencies.

```
type CompleteCoAlg[F[_], S] = OpenCoAlg[F, F, S]
def close[F[_], S](co: CompleteCoAlg[F, S]): CoAlg[F, S] =
  fix[CoAlg[F, S]]{self ⇒ state ⇒ co[S](id)(self)(state)}
def unfold[F[_]: Functor, S](co: CompleteCoAlg[F, S]): S ⇒ Fix[F] =
  state ⇒ new Fix[F]{def out = map(close(co)(state))(unfold(co))}
```

(c) Closing and unfolding complete coalgebras.

```
val SkipC = new OpenCoAlg[(Counter_F With_F Skip_F)_#Apply, Skip_F, Unit] {
  def apply[S](priv: Lens[S, Unit]) = self ⇒ state ⇒ new Skip_F[S] {
    def skip = self.apply(self.apply(state).inc).inc
  }
}
```

(d) Coalgebraic implementation of the skipping counter.

Figure 3.9: *Core Encoding Part I*: coalgebra composition. Expressing dependencies between separate coalgebra definitions.

containing the *println* statement, illustrates method access on the constructed object and will yield 3 when executed.

val *both* = *compose* (*Counter*, *SkipC*)
val *c* = *unfold* (*both*, (0, ()))
println (*c.out.skip.out.inc.out.get*)

In this section we have seen how coalgebras implementing interface functors can be defined in separation, articulating their dependencies to other coalgebras. To this end, we introduced the notion of *open coalgebras* expecting a late bound self-reference as first argument. An instance of *OpenCoAlg* thus is not complete in general. *OpenCoAlg* shares this property with the incomplete classes as introduced by Bettini et al. (2004). In contrast, as opposed to open coalgebras, where only the subset of complete coalgebras can be instantiated, incomplete classes can always be instantiated to incomplete objects, which than in turn can be composed with complete objects, only. However, (Bettini et al., 2004) lacks a mechanism of composing several incomplete classes to a complete class, which is possible with self-type annotations. Open coalgebras also can be compared to open mixin components in the denotational model of inheritance by Oliveira et al. (2012), based on the denotational semantics presented by Cook and Palsberg (1989). Both, the coalgebraic encoding presented here, as well as the denotational model use open recursion as primary means for modular refinement. The model of inheritance defines composition for mixin components which can be arbitrary functions. Since coalgebras are functions $S \Rightarrow F\,[S]$ the coalgebraic encoding can be seen as an instance of the denotational model. However, in (Oliveira et al., 2012) mixins only describe a refinement of the base implementation. Hence, the fine grained (nominal) support for specification of requirements as implemented by open coalgebras is missing. The encoding presented in this section is probably closer to open generalized object algebras in (Oliveira et al., 2013). However, Oliveira et al. use self-references to modularize objects algebras (compositional algorithms, defined on different variants) while in this section we have seen how to modularize interface definitions of one object.

3.4 Dynamically Extending Objects

The last section showed how to express dependencies between separately defined coalgebras by means of *OpenCoAlg*. For Scala programmers, this might not seem like an exciting new feature. Self-type annotations already fulfill a similar purpose in Scala. However, as opposed to traits, coalgebras are first-class values in our encoding. They can be passed as arguments and they can be stored in data structures. This allows a dynamic selection of the implementation of an object at runtime *before* the object is instantiated.

While dynamic coalgebra composition is an important step-stone, let us reemphasize the ultimate goal of our encoding: *dynamic specialization of objects*. Dynamic specialization means that objects can be extended with new operations and modified behavior after their construction. Up to now, coalgebras can be defined as implementations of interfaces and objects can be instantiated by unfolding complete coalgebras. However, after an object has been created the implementing coalgebra as well as the type of the state are hidden behind the fixed point $Fix[F]$. This forbids to exchange the coalgebraic implementation underlying an object of type $Fix\,[F]$ from the outside, hindering the addition of new operations. So how can we possibly modify the encoding to also support dynamic extension of objects?

In order to answer this question, we have to call to mind how objects are modeled

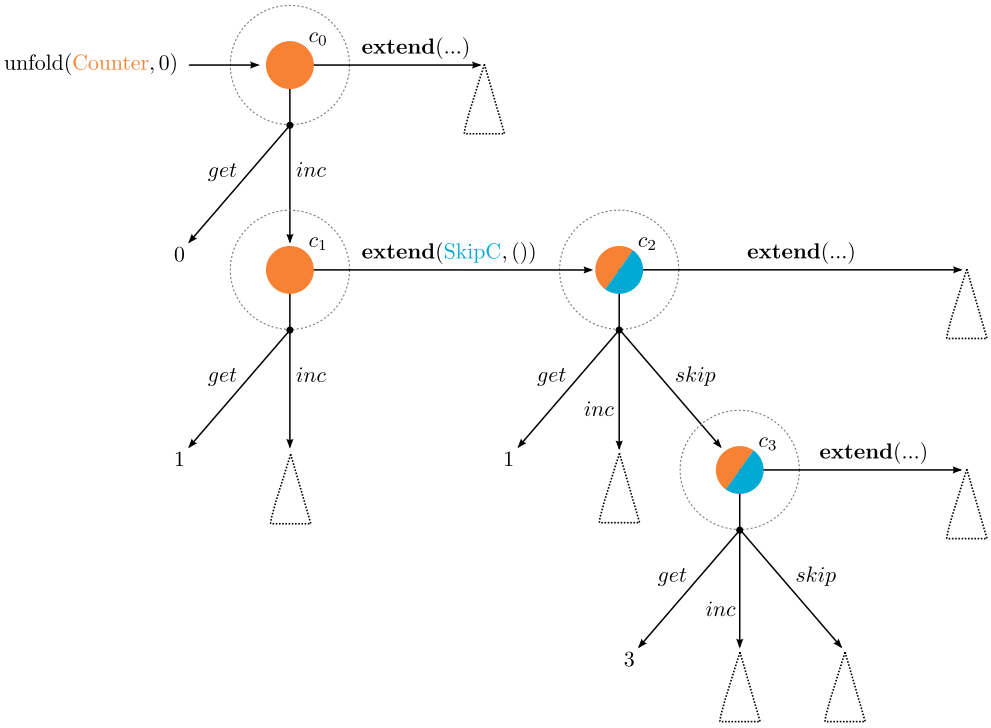


Figure 3.10: Adding extension points to the infinite tree of observations – the terminal co-algebra.

in a coalgebraic encoding. In Figure 3.1c we have seen the infinite tree of observations, a visualization of the terminal coalgebra for the interface endofunctor, representing an object as the greatest fixed point of that interface. Each possible future state of the object is contained in the tree as a node derived by a finite number of observations from the initial state. Every node contains child-nodes for every possible observation that are implemented by performing the next layer of unfolding using the state represented by the node itself.

Importantly, every layer of unfolding is performed lazily only before the next observation is made – each time offering the chance for a refinement of the underlying coalgebra. It is this observation, that equips us with the tool to dynamically specialize the implementation of an object for the next layer of unfolding. In the remainder of this section, we will see how a modification of the fixed point *Fix* allows inserting extension points at every node of the observational tree.

The idea of extensible terminal coalgebras is illustrated by example in Figure 3.10. For this purpose, the tree of observations is shown for the following example program[10]:

$$\begin{aligned}
&\textbf{val } c_0 = \mathit{unfold}\,(\mathit{Counter}, 0)\\
&\textbf{val } c_1 = c_0.\mathit{out}.\mathit{inc}\\
&\textbf{val } c_2 = c_1.\textbf{extend}\,(\mathit{SkipC}, ())\\
&\textbf{val } c_3 = c_2.\mathit{out}.\mathit{skip}
\end{aligned}$$

Unfolding the *Counter* coalgebra with an initial state 0 results in an object of type $\mathit{Fix}[\mathit{Counter}_F]$, depicted by node c_0. Compared to the original model in Figure 3.1c, the extended model now offers two different actions on an object.

Firstly, as before, the observations described by the corresponding functor can be made. To this end, a call to *out* triggers one additional unfolding operation yielding

[10]Here the method **extend** is displayed as keyword to highlight its importance in our encoding.

an instance of $Counter_F[Fix[Counter_F]]$. This way the successor state c_1 is reached by first unfolding one layer and then calling the observation *inc* on the resulting structure. The process of unfolding that is necessary to make observations is depicted by the dotted circular boundary.

Secondly, as a new operation, the underlying coalgebraic implementation can be extended. To this end, an additional coalgebra that might be defined over a different functor, can be internally composed with the original coalgebra to result in an object that allows observations provided by both coalgebras.

In this sense, state c_2 is special. It is reached by invoking **extend** on the existing object c_1 with the coalgebra *SkipC*, adding an implementation for the $Skip_F$ interface. As a result, after the next call to *out* the available observations on c_2 got augmented with the method *skip*. The extended terminal coalgebra (represented by the subtree rooted at c_2) is defined over a different interface endofunctor than the coalgebras represented by the subtrees rooted at c_0 and c_1. While the latter two are $Counter_F$-coalgebras, the one at c_2 is a ($Counter_F$ **with** $Skip_F$)-coalgebra.

Allowing the terminal coalgebra at every level of unfolding to change the functor it is defined over can be considered the major technical novelty introduced by this thesis.

3.4.1 Semantics of **extend**

While the example in Figure 3.10 provided some intuition about the operational behavior of dynamic specialization we will now capture the semantics of **extend** more formally.

For our purpose, let uppercase letters (A, B, ...) refer to coalgebras and let be the state s be indexed by the corresponding coalgebra. We define dynamic specialization in obj.**extend** by the following equation:

$$\text{unfold}(A, s_A)\ \ \textbf{extend}(B, s_B) \stackrel{\text{def}}{=} \text{unfold}(\text{compose}(A, B), (s_A, s_B)) \tag{3.1}$$

Representing an object by the unfolding of its implementing coalgebra A and the current state s_A, the extension with an additional coalgebra B and the corresponding state s_B is defined by the unfolding of the composed coalgebras applied to the tupled state. The composition of coalgebras here refers to the operation *compose* as defined in Figure 3.9.

In the case that state modifying observations have been triggered between instantiation and extension, those are reduced first. This is illustrated by the following example:

$$\begin{aligned}\text{unfold}(A, s_A).obs\ \ \textbf{extend}(B, s_B) &= \text{unfold}(A, s'_A)\ \ \textbf{extend}(B, s_B) \\ &= \text{unfold}(\text{compose}(A, B), (s'_A, s_B))\end{aligned}$$

Here s'_A is the the state resulting from the triggered observation *obs*, given by $A(s_A).obs$.

Associativity. We can notice that composition of coalgebras is associative:

$$\text{compose}((\text{compose}(A, B,)), C) = \text{compose}(A, (\text{compose}(B, C)))$$

We will informally argue, why this is the case by considering three aspects of the semantics of the resulting object: *a*) the set of methods in the interface as reflected by the type, *b*) the defined behavior of the methods, *c*) the binding of self-references.

– The composition operator creates a coalgebra implementing the intersection of the original functor interfaces. An instance created by unfolding the resulting coalgebra hence implements both interfaces – composition is monotonic. As a result, the order in which composition is performed has no influence on the set of implemented methods.

– When composing two coalgebras, the latter can override methods of the first. This is necessary to refine behavior of existing methods in extensions. Hence, if C overrides a method of the composition of A and B, then it will also override a method in B and subsequently in A. The same is true for the opposite direction.

– Unfolding a composed coalgebra will always bind the self-reference to the overall composite. Thus, there is the order of composition has no effect on the binding of self-references.

We conclude that *compose* is associative.

Commutativity. However, both **extend** and *compose* are not commutative. While the same argument for the set of methods and the binding of the self-refences would still hold, the semantics of overriding depends on the order of the coalgebras and thus is non-commutative. For similar reasons there is no distributivity of **extend** and triggered observations.

Limitations of equational reasoning. Equation 3.1 defines **extend** in terms of *compose*, thus it is always safe by definition to replace a program matching the left-hand-side of the equation with the corresponding program on the right-hand-side of the equation. This can be interpreted as: Every dynamically specialized object could also be an instance of dynamically composed classes. However, the opposite is not the case. In our encoding it is not possible to instantiate every class (represented by a coalgebra) to specialize it later.

A simple example in Scala can illustrate this limitation. Assuming that a trait A has abstract members that are only implemented by a different trait B. We can instantiate A **with** B but not A alone since it has unimplemented members In terms of our coalgebraic encoding this equivalent to the restriction that only complete coalgebras might be unfolded. While we always know that the result of extending a complete coalgebra will be a complete coalgebra as well, it is not the case that the left-hand-side A of every complete composition *compose* (A, B) is complete as well.

3.4.2 Implementation of **extend**

To account for the two different actions *out* and **extend** that can be performed on a terminal coalgebra we need to adapt the interface of *Fix*. Figure 3.11 shows the modified definition. *Fix* has been augmented with the new method **extend** which takes a open coalgebra co_2 and the second state component $state_2$ necessary to unfold co_2. In addition to the two arguments it also expects evidence that co_2 is defined over a functor G. The result of **extend** is the terminal coalgebra defined over the intersection of the two functors F and G. Figure 3.11 also shows the implementation of the method **extend** as part of the modified function *unfold*[11]. Reusing the definition of *compose* from Figure 3.9b the implementation just amounts to composing the two coalgebras and recursively unfolding the result with the tupled state. Since *unfold*

[11]For ease of presentation, the handling of type-tags is omitted. The acquisition and threading of type-tags through the methods calls is purely technical and does neither contribute to the semantics of *unfold* nor to that of **extend**.

```
trait Fix[F[_]] {
  def out: F[Fix[F]]
  def extend[G[_]: Functor, S2](
    co2: OpenCoAlg[(F With_F G)_#Apply, G, S2],
    state: S2): Fix[(F With_F G)_#Apply]
}
def unfold[F[_]: Functor, S1](co1: CompleteCoAlg[F, S1]): S1 ⇒ Fix[F] =
  state1 ⇒ new Fix[F] {
    def out = map(close(co1)(state1))(unfold(co1))
    def extend[G[_]: Functor, S2](
      co2: OpenCoAlg[(F With_F G)_#Apply, G, S2],
      state2: S2): Fix[(F With_F G)_#Apply] =
      unfold(compose(co1, co2))(merge[F, G])((state1, state2))
  }
```

Figure 3.11: *Core Encoding Part II*: Allowing later extension of objects by augmenting the fixed point construction. Differences to the previous versions of *Fix* in Figure 3.2c and *unfold* in Figure 3.9c are highlighted in *gray*

requires evidence that the passed coalgebra is defined over a functor, a merged functor instance is created by calling $merge[F, G]$. The merged functor instance is defined by point-wise applying *map* defined in $Functor[F]$ and $Functor[G]$ to the intersection $F[A]$ **with** $G[A]$ before composing the results using an instance of F $\mathbf{With_F}$ G.

The encoding of the fixed point using a trait (or case class) *Fix*, often is just a necessary workaround in languages like Scala that only support isorecursion but not equirecursion. In Haskell a **newtype** declaration is often used instead, erasing the distinction between both sides of the equation after type-checking:

$$\textbf{newtype}\ Fix\ f = In\ (f\ (Fix\ f))$$

In this thesis, having a manifestation of the fixed point at every level of recursion is essential. It allows a covariant refinement of the interface endofunctor F that *Fix* is defined over. In addition, the requirement that methods can be overridden and redefined is implemented by the right biased nature of the object composition operator *mix*.

As noted earlier, the type of the private state S_1 and the original coalgebraic implementation are hidden after unfolding to the greatest fixed point $Fix[F]$. However, both the state as well as the coalgebraic implementation are *stored in the closure* of **extend** and hence enable a later retroactive refinement.

In this section we have seen how the greatest fixed point (the terminal coalgebra) can be augmented to account for extension points at every layer of unfolding. The the definition of **extend** in Equation 3.1 illustrates that we reduce the problem of dynamically extending objects to the problem of dynamically composing coalgebras. The technical implementation of this process is twofold: Composition of the internal state which is passed to the component coalgebras and composition of the interface functors F and G, building on the results of Section 3.3 and Section 3.2, respectively.

Neither the type of the state nor the original coalgebra are revealed. They are stored in the closure of **extend**. Calling **extend** with another coalgebra, can be seen as instantiating the extension point in the observation tree with the result of unfolding the composed coalgebra to the observational tree with a (possibly) different shape.

3.4.3 Subtyping

While we have modeled a form of dynamic inheritance in the last subsections, subtyping has yet to be discussed. From the tight coupling between inheritance and subtyping as known from object oriented languages, one might assume that an object created from a $Counter_F$-coalgebra, extended with a $Skip_F$-coalgebra can be used both as a *Counter*-object and a *Skip*-object since it implements both interfaces.

val $c = unfold\,(Counter, 0).$**extend**$\,(SkipC, ())$
val $simpleCounter$: $Fix[Counter_F] = c$ // *Type Error*

However, the above assignment fails with a type error in Scala. The type of c, namely the fixed point of the intersection functor $Fix[Counter_F$ **with** $Skip_F]$ is not a subtype of the fixed point of one component $Fix\,[Counter_F]$. At the same time, one might assume that a coalgebra like *Counter* with a self-type of $Counter_F$ can be used to repeatedly extend c which does implement both $Counter_F$ and $Skip_F$. This is useful for instance to reset the counter to zero.

c.**extend**$\,(Counter, 0)$ // *Type Error*

Surprisingly the above line of code also fails with a type error, even though the requirements of *Counter* are overly fulfilled by c.

There is a simple technical explanation for both failures. Both, coalgebras and the objects instantiated from them are parametric in their type. The type parameters of coalgebras are used to articulate dependencies and the type parameter of the fixed point *Fix* expresses the implemented interface. However, all definitions of coalgebras and fixed points we have seen so far missed variance annotations (mainly for ease of presentation) at their type parameters and so the Scala compiler infers the type constructors as *invariant* in their type parameters.

The remainder of this subsection is a systematic overview how to add variance annotations to the previous definitions.

Variance annotations allow to describe the polarity of type constructor arguments. Interpreting a type constructor of kind $* \to *$ as endofunctor F over the category **Set** than the type constructor is said to be *covariant* if for each morphism $A \Rightarrow B$ it associates a morphism $F[A] \Rightarrow F[B]$, that is, in the interpretation of types: "if A is a subtype of B, then also $F[A]$ is a subtype of $F[B]$". On the other hand, contravariant functors reverse the order of the morphism and thus associate each morphism $A \Rightarrow B$ a morphism $F[B] \Rightarrow F[A]$ with the corresponding interpretation on types that "if A is a subtype of B, then also $F\,[B]$ is a subtype of $F\,[A]$". In Scala, variance can be marked at the definition site using the annotation + for covariant (or positive) type parameters and − for contravariant (or negative) type parameters. If no annotation is present the type parameter is assumed to be invariant.

We will begin with restoring subtyping on objects modeled by the greatest fixed point *Fix*. Since for interface functors A_F and B_F the fixed point of their intersection functor $Fix\,[A_F$ **with** $B_F]$ should be a subtype of both $Fix\,[A_F]$ and $Fix\,[B_F]$ the type parameter F of $Fix[+F[\_]]$ should be marked as covariant. Furthermore, since unfolding *Fix* one layer by calling *out* results in $F\,[Fix\,[F]]$ and the type parameter of F is invariant, the covariant type F occurs in an invariant position, as part of its own invariant type parameter. This problem can be solved by also marking F to be covariant in its type parameter. Thus the annotated definition of *Fix* reads:

trait $Fix[+F[+\_]]\,\{...\}$

The method **extend** in trait *Fix* has the result type $Fix\,[(F\ \mathbf{With_F}\ G)_{\#\mathrm{Apply}}]$. This forces us to mark the type parameter of $(\mathbf{With_F})_{\#\mathrm{Apply}}$ and in consequence the

arguments of the type constructor parameters to $\mathbf{With_F}$ also as covariant, leaving us with the updated definition of $\mathbf{With_F}$.

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

The above changes also impose the additional requirement that all interface endofunctors must be covariant. This is the case for all examples we have seen so far. However, it prevents the self-type of a functor to be used for method arguments, since those are negative occurrences.

Of course all methods that are parametrized over an interface endofunctor like *unfold* and *compose* also have to be adapted to account for the covariance of the functor. The full source code, containing all necessary variance annotations can be found in our online distribution.

In addition to the result type of the method **extend**, type parameter F also occurs as component of the self-type requirement on *OpenCoAlg*. The resulting variance conflict coincides with the desired subtyping relationship of open coalgebras. Open coalgebras should be contravariant in their requirements and covariant in their provided interface.

Updating the definition of *OpenCoAlg* thus amounts to:

```
trait OpenCoAlg[-Self[+X] <: Prov[X], +Prov[+_], S] {...}
```

This concludes the development of an coalgebraic encoding that allows the modular definition of objects with regard to temporal decomposition criteria. In particular, dynamic specialization of objects is supported by the means of inserting extension points at every layer of unfolding. Adhering to the principle of information hiding, the extension of objects does neither reveal the internal state nor the original coalgebraic implementation. Instead, the state as well as the implementation are stored in the closure of *Fix* – only to be composed with future extensions by means of the composition operator on coalgebras: *compose*. To allow a consistent handling of private state in presence of multiple participating coalgebras, Lenses are used for access and modification of the corresponding state-slices. Finally, the subtyping relationship between instantiated objects as well as implementing coalgebraic fragments are restored, guaranteeing that extension of objects is a safe monotonic operation.

Chapter 4

Possible Extensions

After having developed the core of the encoding over the course of the last chapter, the current chapter will discuss two extensions to the core. The first extension allows explicitly calling overridden methods of the extended base object. The second extension builds on the first and additionally implements *selective open recursion*, a feature less wide-spread.

Both extensions are summarized by example in Figure 4.1. The depicted arrows have a single coalgebra as the source and another (possibly composed) coalgebra as the target. An arrow hence describes the binding of the labeled reference for a given coalgebra.

4.1 Extension 1: Referencing the Base

The core introduced in the last chapter allows dynamically specializing objects by extending the interface functor the terminal coalgebra is defined over. One special case is when the interface of the extension has a non empty intersection with the interface of the extended base. Since *compose* as introduced in Section 3.3 uses the right-biased operator *mix*, the implementations in the extension will always shadow the ones of the base. In object oriented language terminology this case would usually be referred to as *override* of the intersection. However, often it is desired to not only override implementations but to refine them. To this end, the common idiom is to allow accessing overridden implementations via some special prefix – in many languages this prefix is called **super**[1].

Following the implementation pattern of late bound self-references, we can add an additional reference *base* to open coalgebras. This change becomes visible in Figure 4.2a. *OpenCoAlg* is redefined and a new contravariant type parameter *Base* is added to the interface. At the same time coalgebras defined in the body of *apply* can depend on the base coalgebra which is passed as additional argument.

An example that illustrates the usage of the newly defined *OpenCoAlg* can be found in Figure 4.2d: An instrumented counter that prints a message every time the original counter is incremented. The implementation overrides both methods *get* and *inc*, but acts as a proxy by forwarding them to the base coalgebra. Only in the *inc* case the message is printed before invoking the base coalgebra with the current state.

The semantics of the base reference is defined by the function *compose* in Figure 4.2b. In the signature of *compose* it becomes visible that co_2 can depend on both $Prov_1$ as well as $Base_1$. This allows accessing overridden methods that are not part of the interface of the immediate ancestor but defined somewhere in the transitive chain of parents. Most importantly, the *base* reference of the composite is passed to the first coalgebra co_1, while co_1 is passed (as part of *left*) to co_2. This assures a linearization of the base references. The coalgebra returned by *compose* requires $Base_1$ as type of

[1] To be precise, **super** has a different semantics than base-references. The receiver of calls to **super** can be resolved statically. It is the superclass. In our case the *base* reference is bound dynamically at extension time.

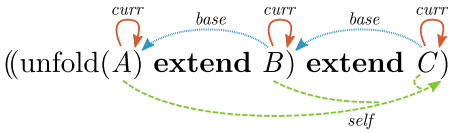


Figure 4.1: Example reference structure using both extensions. The *base* reference (Extension 1) points to the implementing coalgebra of the previously extended object, while the *curr* reference (Extension 2) points to the coalgebra of the object at the time of extension.

a future base to be composed with.

As opposed to the type parameter *Self* which most likely specifies an interesting interface (after all being forced to by $Self <: Prov$), the type of *Base* might impose no requirements. If base would be a simple type of kind $*$, we could just use the Scala built-in type *Any* for this purpose. Since *Base* is a type constructor we introduce the corresponding top element of the lattice for type constructors of kind $* \to *$ as:

$$\textbf{type}\ Any_F[+\_] = AnyRef$$
$$\textbf{def}\ Any_F[S] = (s: S) \Rightarrow \textbf{new}\ Any_F[S]\ \{\}$$

The equally named method Any_F is a convenience function to define dummy coalgebras of type $S \Rightarrow Any_F[S]$. Open coalgebras without requirements regarding their base can thus be written as: $OpenCoAlg[Any_F, ..., ..., ...]$. The contravariance annotation assures that these coalgebras can also be composed with other coalgebras more specific in their provided interface. An example of a usage of Any_F for this purpose can be found in Figure 4.2c. To recall, a complete coalgebra meets its own requirements and does not depend on any further coalgebraic implementations. Thus the type parameter of *Base* is instantiated with Any_F. Hence, in the implementation of *close* the dummy coalgebra Any_F is passed as base reference.

Having given the differences of the necessary definitions, we know can return to the example of the instrumented counter. Given any instance of a counter

$$\textbf{val}\ c: Fix[Counter_F] = ...$$

the refinement can be applied by extending the counter.

$$\textbf{val}\ instr = c.\textbf{extend}\,(InstrCounter, ())$$

Thus calling *instr.out.inc.out.inc.out.get* will print `"Called inc!"` twice and will yield a counter value increased two times.

4.2 Extension 2: Selective Open Recursion

In object oriented programming languages with support for overriding and late binding sometimes refinements made in a subclass make assumptions about implementation details of the base class which are not *explicitly* articulated. A change of the base class implementation, while preserving compliance to the interface, thus may violate these implicit assumptions. This problem is called the *fragile base class problem* (Mikhajlov and Sekerinski, 1998).

Aldrich and Donnelly (2004) identify open recursion as the root of the fragile base class problem: Open recursion allows subclasses to intercept calls to methods made within the base-class, analyzing the call structure and thereby violating the principle of information hiding.

```
trait OpenCoAlg[ −Base[+_], − Self[+X] <: Prov[X], +Prov[+_], S] {
  def apply[T] (priv: Lens[T, S]):
    CoAlg[Base, T] ⇒ CoAlg[Self, T] ⇒ CoAlg[Prov, T]
}
```

(a) Adding a type parameter *Base* to express requirements on the parent implementation.

```
def compose[
    Base1[+_], Self1[+X] <: Prov1[X], Prov1[+_], S1,
    Self2[+X] <: Prov2[X], Prov2[+_], S2] (
    co1: OpenCoAlg[ Base1, Self1, Prov1, S1],
    co2: OpenCoAlg[ (Base1 With_F Prov1)_#Apply, Self2, Prov2, S2]) =

  new OpenCoAlg[ Base1, (Self1 With_F Self2)_#Apply, (Prov1 With_F Prov2)_#Apply, (S1, S2)] {
    def apply[T] (priv: Lens[T, (S1, S2)]) = base ⇒ self ⇒ state ⇒ {
      val left = (s: T) ⇒ mix[Base1[T], Prov1[T]] (
        base (s),
        co1[T] (first compose priv) (base) (self) (s)
      )
      val right = co2[T] (second compose priv) (left) (self)
      mix[Prov1[T], Prov2[T]] (left (state), right (state))
    }
  }
```

(b) The left coalgebra is passed as reference *base* to the right coalgebra.

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

def close[F[+_], S] (co: CompleteCoAlg[F, S]): CoAlg[F, S] =
  fix[CoAlg[F, S]] { self ⇒ state ⇒ co[S] (id) (Any_F[S]) (self) (state) }

trait Fix[+F[+_]] {
  def out: F[Fix[F]]
  def extend[G[+_]: Functor, S2] (
    co2: OpenCoAlg[ F, (F With_F G)_#Apply, G, S2],
    state: S2): Fix[(F With_F G)_#Apply]
}
def unfold[F[+_]: Functor, S1] (co1: CompleteCoAlg[F, S1]): S1 ⇒ Fix[F] =
  state1 ⇒ new Fix[F] {
    def out = map (close (co1) (state1)) (unfold (co1))
    def extend[G[+_]: Functor, S2] (
      co2: OpenCoAlg[ F, (F With_F G)_#Apply, G, S2],
      state2: S2): Fix[(F With_F G)_#Apply] =
      unfold (compose (co1, co2)) (merge[F, G]) ((state1, state2))
  }
```

(c) To close the self-references of a complete coalgebra, a dummy object is passed as base-reference.

```
object InstrCounter extends OpenCoAlg[Counter_F, Counter_F, Counter_F, Unit] {
  def apply[T] (priv: Lens[T, Unit]) = base ⇒ self ⇒ state ⇒ new Counter_F[T] {
    def get = base (state).get
    def inc = { println ("Called inc!"); base (state).inc }
  }
}
```

(d) An instrumented counter coalgebra – it requires *Base* to implement the interface $Counter_F$.

Figure 4.2: *Extension 1*: Allowing references to the overridden base object. Changes, compared to the core encoding are highlighted in *gray*.

Using the *base*-extension introduced in Section 4.1 we might implement the canonical example (Steyaert et al., 1996; Aldrich and Donnelly, 2004) of an instrumented set, that maintains the count of added elements as its state by overriding *add* and *addAll* as follows:

```
...
def add (obj: Any) = {
  val s = base (state).add (obj);
  priv.set (s, priv.get (s) + 1)
}
def addAll (objs: Seq[Any]) = {
  val s = base (state).addAll (objs);
  priv.set (s, priv.get (s) + objs.size)
}
...
```

This implementation makes the implicit assumption that *add* and *addAll* are defined independent of each other. If the authors of the base class now refactor the code and implement *addAll* in terms of *add*, elements will suddenly be counted twice.

A possible solution to this problem is to articulate more implementation details, such that they are not hidden anymore. However, this defeats the purpose of the principle of information hiding by allowing dependencies on information that is likely to change (Parnas, 1972).

Another solution is introduced by Aldrich and Donnelly (2004). The authors suggest to only selectively enable open recursion when it is necessary by means of a modifier `open`. Hence, only for those methods that are marked as open, the invocation target will be bound late.

While this declaration-site modifier cannot be supported directly, the implementation of use-site selective open recursion is straight-forward in our encoding. To allow the user to select whether a *method call* should be bound late or not, both implementations the late bound one as well as the current can be passed as arguments. Building on the *base* extensions, this results in the definition of *OpenCoAlg* in Figure 4.3a.

For a user of the encoding the only visible difference is the duplicated argument of type $CoAlg[Self, T]$ passed to open coalgebras. The implementation of the *compose* operator in Figure 4.3b then specifies the semantic difference between the two arguments *curr* and *self*.

With every applied extension, the binding of the self-reference changes to the outermost coalgebra. Thus, in Figure 4.1 the self-reference of all three coalgebras point to the same target.

The *curr* reference of a coalgebra instead always points to the self-reference at the *time of extension* with that particular coalgebra. In consequence, for the coalgebra that has been added last there is no difference between the references *curr* and *self*. This becomes visible with the coalgebra C in Figure 4.3b and in the implementation of *close* in Figure 4.3c.

The "freezing" of the self-reference to the reference *curr* at the time of extension then is modeled by the fixed point construction in Figure 4.3b.

Using this extension we now can define a set (Figure 4.3d) that behaves robust faced with additional implicit assumptions about the internal call structure. The method *addAll* now is implemented in terms of the method *add* defined in the current coalgebra. Even if a subclass (an extension) like the counting set above overrides *add*, calls to *curr* (*state*).*add* will not be bound late and thus will not invoke the overridden method.

```
trait OpenCoAlg[−Base[+_], −Self[+X] <: Prov[X]], +Prov[+_], S] {
  def apply[T](priv: Lens[T, S]):
    CoAlg[Base, T] ⇒ CoAlg[Self, T] ⇒ CoAlg[Self, T] ⇒ CoAlg[Prov, T]
}
```

(a) An additional parameter in *apply* allows distinguishing the current and the late bound self-reference.

```
def compose[
    Base1[+_], Self1[+X] <: Prov1[X], Prov1[+_], S1,
    Self2[+X] <: Prov2[X], Prov2[+_], S2](
    co1: OpenCoAlg[Base1, Self1, Prov1, S1],
    co2: OpenCoAlg[(Base1 With_F Prov1)#Apply, Self2, Prov2, S2]) =
  new OpenCoAlg[Base1, (Self1 With_F Self2)#Apply, (Prov1 With_F Prov2)#Apply, (S1, S2)] {
    def apply[T](priv: Lens[T, (S1, S2)]) = base ⇒ curr ⇒ self ⇒ state ⇒ {
      val left = (s: T) ⇒ mix[Base1[T], Prov1[T]](
        base(s),
        fix[CoAlg[F, T]] {
          curr ⇒ co1[T](first compose priv)(base)(curr)(self)(s)
        }
      )
      val right = co2[T](second compose priv)(left)(curr)(self)
      mix[Prov1[T], Prov2[T]](left(state), right(state))
    }
  }
```

(b) The left-hand-side coalgebra of a composition is passed to itself as the current self-reference.

```
def close[F[+_], S](co: CompleteCoAlg[F, S]): CoAlg[F, S] =
  fix[CoAlg[F, S]]{self ⇒ state ⇒ co[S](id)(self)(self)(state)}
```

(c) For the outermost coalgebra the late bound self-reference and the current-reference are the same.

```
object Set extends OpenCoAlg[Any_F, Set_F, Set_F, ...] {
  def apply[S](priv: Lens[S, ...]) =
    base ⇒ curr ⇒ self ⇒ state ⇒ new Set_F[S] {
      def add(obj: Any) = {...}
      def addAll(objs: Seq[Any]) = objs.foldLeft(state) {
        case(state, obj) ⇒ curr(state).add(obj)
      }
    }
}
```

(d) An implementation of a *Set* that is immune to the fragile base class problem.

Figure 4.3: *Extension 2*: Implementing selective open recursion by passing references to both the current as well as the late bound co-algebraic implementation. Changes, compared to the *base*-extension in Figure 4.2 are highlighted in *gray*.

In this chapter we have seen two different extensions to the core encoding. While the first one is fairly standard and has been used in many experiments with the encoding (also see Section 5.1), the second extension shows that also non standard extensions like selective open recursion are relatively easy to add to the encoding.

Chapter 5

Evaluation and Discussion

This chapter evaluates the encoding elaborated in previous chapters by presenting and discussing three use case examples in Section 5.1; *a*) a canonical example often used to present the decorator pattern (Gamma et al., 1994), *b*) an example that has been used to introduce dynamic specialization in (Ernst, 1999), and *c*) a more realistic use case that translates parts of the OpenJDK implementation of stream writers to our encoding. Section 5.2 then discusses the encoding in general, comparing it to related work.

5.1 Usage Examples

Throughout the previous chapters we have encountered only a few examples of how the encoding presented in this paper can be used. The running example we have seen was a modularized variant of building counters with different sets of functionalities. Each feature required us to improve the encoding in order to be able to support implementing it. Purely functional encodings of these running examples can also be found in (Pierce, 2002, Chapter 32). However, where these encodings target at restoring the expressiveness of class based object-oriented languages, the encoding presented in this thesis additionally supports the incremental construction of objects where every stage of construction represents a valid object. At the end of Chapter 32, one exercise is left to the reader by Benjamin Pierce: "Define a subclass of *InstrCounter* that adds *backup* and *reset* methods". In our framework we can not only define such a subclass by simply composing *InstrCounter* with *BackupCounter*. We can also retrofit a counter object incrementally with both, instrumentation as well as backup functionality.

In addition to the examples from "Types and Programming Languages" that served as a running example we also translated two other small use cases from literature as well as a medium size case study which is closer to real world problems. In the remainder of this section we will briefly discuss these three usage examples.

5.1.1 Window Decorators

The decorator pattern is a wide-spread programming technique for dynamically changing program behavior and enriching objects with new functionality. As discussed in Section 2.2.3, the decorator pattern also comes with a number of deficiencies that are targeted by the encoding in this thesis, namely (a) composition of independently defined decorators that each add to the interface and (b) late binding of methods. To facilitate comparison of the decorator pattern with our solution we translated the canonical example from (Gamma et al., 1994) to the encoding presented in this thesis. Since decorators enable refinement of method implementation by forwarding to the decorated object we chose the core encoding enhanced with the extension for referencing the base as introduced in Section 4.1.

The example is set in the context of graphical user interface programming. A *TextView* is a visual component that allows rendering text in a graphical pane. The two decorators *ScrollDecorator* and *BorderDecorator* both allow decorating visual components while the first also adds the method *scrollTo* and the latter adds the method *drawBorder*.

In comparison with the decorator based implementation it becomes visible that the interfaces *TextView*, *ScrollDecorator* and *BorderDecorator* can be specified completely independent of each other. The inheritance structure, with all three interfaces being a subtype of a common superclass, as required by the decorator pattern is not mandatory with our encoding. In general, dependencies are not longer necessarily being expressed via the interfaces (the endofunctors) but are now articulated over the type parameters of an coalgebraic implementation. This allows delaying the check whether requirements are satisfied to coalgebra composition time.

As seen in the previous sections, coalgebra composition can model both static trait composition but also dynamic specialization of objects. Although the three interfaces can now be described in isolation, objects that feature all three interfaces can be constructed by either

$$\begin{aligned}&\textbf{val } o_1 = \mathit{unfold}\,(\\&\quad \mathit{compose}\,(\mathit{compose}\,(\mathit{TextView},\, \mathit{ScrollDecorator}),\, \mathit{BorderDecorator}),\\&\quad ((\texttt{"..."}, (5, 0)), ()))\end{aligned}$$

modeling static composition, or alternatively by

$$\begin{aligned}&\textbf{val } o_2 = \mathit{unfold}\,(\mathit{TextView},\, \texttt{"..."})\\&\quad .\textbf{extend}\,(\mathit{ScrollDecorator}, (5, 0))\\&\quad .\textbf{extend}\,(\mathit{BorderDecorator}, ())\end{aligned}$$

modeling dynamic specialization of objects. Both objects o_1 and o_2 expose the same observable runtime behavior as the variant based on the decorator pattern.

This example also shows how our encoding meets the transparency requirement, while the decorator pattern does not. In our case the type of the composites o_1 and o_2 is the intersection type of all three interfaces, while the variant using the decorator pattern would have type *BorderDecorator*. Hence, method *scrollTo* can only be accessed in our encoding but not in the decorator pattern based solution. Our encoding thus behaves fully transparent.

5.1.2 Incremental Object Creation

Ernst (1999) introduces incremental object creation as a disciplined way to use dynamic specialization. Objects that are used across module boundaries might contain parts that are specific to certain modules. In Scala, using traits those parts can be factored out into the corresponding module. However, the *construction* of objects remains both static and monolithic. Often the responsibility for object construction is shifted to a top-level module that also performs the wiring of all modules.

In contrast, with dynamic specialization even the module that is responsible for the construction of objects can be oblivious of the different parts that are contributing to the object. This has the benefit that each module is aware only of a certain partition of the object, facilitating change of the rest without breaking that particular module. To show that our encoding is sufficiently expressive enough to support incremental object creation we translated the canonical example from (Ernst, 1999, Section 7.3.3) to our encoding.

The example is set in the context of writing software for a large company with multiple departments. The business entity, a *Car*, is uniquely identified by its

registrationNumber. However, each department has its own viewpoint on the car. The finance department views the car as *Property* that can be written off over time, focusing on its *price*. The company-internal car rental department views the car as *Schedulable* with *reserve* (*from*: *Date*, *to*: *Date*) as its business critical method. The central hub is responsible for creating and managing *Car* objects, but is oblivious of the different departments that use its service to get access to cars.

In our translation, each aspect of the car is modeled by an interface and its corresponding coalgebraic implementation. The hub can then be implemented as follows:

```
val carEnhancers = ListBuffer.empty[Car ⇒ Car]
def buildCar (regNum: Int): Car =
  carEnhancers.foldLeft (unfold (Car, regNum)) {
    case (car, enhancer) ⇒ enhancer (car)
  }
```

The other departments can contribute to the car construction process by registering an car enhancer.

```
carEnhancers += {_.extend (Property, 10500)} // finance dep.
carEnhancers += {_.extend (Schedulable, ())}  // rental dep.
```

We also could have designed an enhancer registration method that is parametrized by an *OpenCoAlg* while the dynamic specialization using **extend** is performed inside the hub. This is only possible because coalgebraic implementations of interfaces are available as first-class element in the encoding, whereas traits and classes are second-class in Scala.

After registering the enhancers, every car constructed by the method *buildCar* will also implement *Property* and *Schedulable*. In order to use functionality that is specific to one department we have to pattern match on the specialized type.

```
buildCar (45315).out match {case it: PropertyF[_] ⇒ println (it.price)}
```

This is due to the imprecision in the type of *carEnhancers*. Type information is lost in the result type of *Car* ⇒ *Car* by subsumption. However, the same also holds for the original gbeta implementation.

After all, this style of *Car* construction reduces the dependencies between the departments and the hub to a minimum. Changes within one department do neither affect the other departments nor the central hub. Dynamic specialization and first-class patterns, that is classes as values, enable this style of programming in gbeta. Our encoding brings a similar flexibility to the Scala programming language by allowing retroactive extension of fixed points and representing implementations as coalgebras which are first-class values in Scala. We were able to seamlessly translate the example (Ernst, 1999, Section 7.3.3) to the encoding presented in this thesis, indicating that the encoding is expressive enough to provide good support for incremental object creation.

5.1.3 Stream Writers (OpenJDK)

In order to see how the encoding scales from small examples to more realistically sized programs we reimplemented parts of the OpenJDK6 library[1]. We chose a part of the OpenJDK that is concerned with writing characters to an output stream.

[1]The original source code can be found at `http://hg.openjdk.java.net/jdk6/jdk6/jdk/file/2d585507a41b/src/share/classes/java/io/`

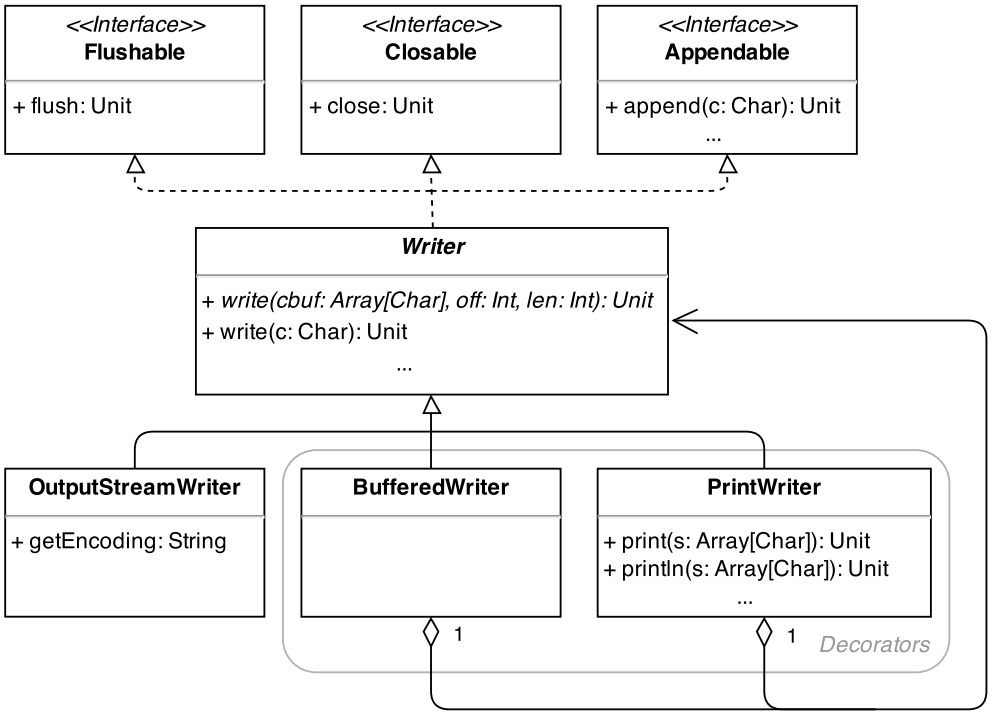


Figure 5.1: UML Class Diagram visualizing the subset of OpenJDK 6 that has been translated to our encoding.

The reason for this choice is that several stream-writers are already implemented using the decorator pattern, facilitating the translation to the framework of dynamic specialization.

The components that have been translated to our encoding are visualized in Figure 5.1. The translated subset consists of *a*) the interfaces *Flushable*, *Closable* and *Appendable*, *b*) the abstract class *Writer* which implements all three interface and *c*) the classes *OutputStreamWriter*, *BufferedWriter* and *PrintWriter*, all three extend *Writer*. The abstract class *Writer* contains one abstract method *write*. All other overloaded variants of *write* are implemented in terms of this abstract method. The class *OutputStreamWriter* extends *Writer* and adds the implementation of the abstract method *write*. Additionally, it contains the method *getEncoding* that is not present in the interface of *Writer*. Both *BufferedWriter* and *PrintWriter* are decorators for *Writer* and thus inherit from *Writer* while at the same time referencing *Writer* to allow forwarding to the decorated instance. The decorator *PrintWriter* also adds methods *print* and *println* for various argument types.

The translated result can be used as follows:

$$\begin{aligned} &\mathit{unfold}\,(\mathit{compose}\,(\mathit{ConcreteWriter},\ \mathit{OutputStreamWriter})) \\ &\quad .\mathbf{extend}\,(\mathit{PrintWriter}, ()) \\ &\quad .\mathbf{extend}\,(\mathit{BufferedWriter},\ \mathit{BufferedWriterState}\,()) \end{aligned}$$

This creates an instance of *OutputStreamWriter* that is then immediately decorated with *PrintWriter* and *BufferedWriter*. We can see that *OutputStreamWriter* is composed with *ConcreteWriter* before being instantiated. Why is this the case? While recovering many features like open-recursion, self-type annotations, referencing the base, and private state we did not specifically address abstract members of a class. The abstract class *Writer* has one abstract method *write*. To implement *Writer* coalgebraically we need to define the function $(s\colon S) \Rightarrow \mathbf{new}\ Writer_F\,[S]\ \{...\}$, which is

not possible due to the abstract method hindering instantiation.

The first solution that may come to mind is to provide a dummy implementation that raises a *MethodImplementationMissing* exception at runtime, if the method is called without being overridden by another coalgebra. However, this gives rise to two possible errors. First, one might simply forget to override the abstract method, leaving clients of the buffered writer with the above exception at runtime. Second, an overriding coalgebra that does implement the abstract method might perform calls to the base reference since the method is present. This again will trigger the above mentioned exception.

An alternative solution is to manually partition the interface of $Writer_F$ into two disjoint interfaces $AbstractWriter_F$ and $ConcreteWriter_F$ according to **abstract** flag being present on a method or not. We can then define $Writer_F$ as:

$$\textbf{type}\ Writer_F[X] = AbstractWriter_F[X]\ \textbf{with}\ ConcreteWriter_F[X]$$

The implementation of *ConcreteWriter* uses $Writer_F$ as self-type, indicating that it also requires the implementation of $AbstractWriter_F$. In our example this is achieved by composition with *OutputStreamWriter*, a coalgebra that implements both $OutputStreamWriter_F$ and $AbstractWriter_F$ and thus fulfills the necessary requirements for being instantiated.

Arguably, this solution imposes additional boilerplate on the translation of the class *Writer*. However, we consider it worthwhile since it recovers the static checks for abstract methods, present in the original.

During the development of this use case example we were faced with yet another problem: The implementation of the method *format* in *PrintWriter* demanded us to create a new instance of *java.util.Formatter*. The constructor of *Formatter* takes an instance of *Appendable* which happens to be implemented by the class *Writer* and thus also by the reference **this** in the original implementation. However, in the encoding as it is, within the coalgebraic implementation we do not have access to **this**. Only the late bound coalgebra is passed as an argument to the implementation. While the encoding might be changed to also pass the fixed point of the self-type $Fix[Self]$ as additional argument to coalgebraic implementations this does not fully solve the issue.

Another problem, orthogonal to the issue of accessing the fixed point, is related with passing references to objects at all. Since our encoding is a purely functional one, state changing method-calls on a reference to an object handed to a client will not be reflected in the state of the original object. Of course the client code could be adapted to also return the object in the new state. In our case this is not possible since *Formatter* is part of *java.util* and thus not under our control. In general, passing references of functional objects to construct a network of collaborating objects is non trivial. This problem is common to all functional encodings of objects but is considered out of scope for this thesis.

Despite the disadvantages of our encoding that came to light during the development of the case study, the translation also shows many advantages over the original implementation.

Not surprisingly, the decorators *BufferedWriter* and *PrintWriter* can be applied in arbitrary order and yet the methods added by *PrintWriter* will always be available on the resulting objects. This is a result that has already been reported for the other examples above.

For the same technical reasons, decorators can be immediately applied on subtypes of classes they have been originally designed for. In the original implementation applying *BufferedWriter* to an instance of *OutputStreamWriter* would hide the method

getEncoding, since the decorator has been designed to only decorate instances of *Writer*.

Late binding of decorated methods is supported without further ado. This allowed us to implement a logger by overriding the abstract method *write*, printing debug information whenever a call is made to *write*. By late binding the debugging printout will also include internal calls within the object's implementation.

Comparing the source code of this example with the source code of the other examples, we can notice that the ratio boilerplate / actual implementation is much better for larger interfaces, such as the one in this example. The boilerplate necessary to define a coalgebraic implementation does only depend on the dependencies that are expressed over the type-parameters of *OpenCoAlg*. Hence, the syntactic impact is much smaller with reasonably sized method bodies. However, we are consciously ignoring the overhead introduced by using a functional encoding at all. This includes threading the state through all method calls and using lenses to project into the state, adding to the overall syntactic weight.

5.2 Discussion

Translating the examples to our encoding showed that it is possible to perform dynamic specialization in Scala without the need for a new compiler or a customized preprocessor. However, the encoding is syntactically verbose compared to proposed language extensions like generic wrappers or incremental objects, that introduce special language constructs targeting at dynamic specialization. We can identify three sources of verbosity:

- Definition site boilerplate. This includes the definition of the interface endofunctors and the scaffold for specifying coalgebraic implementations. Since the transformation from simple interface descriptions is a purely mechanical process, we conjecture that both could be reduced by the use of annotation macros in Scala (Burmako, 2013). In deed, Cai (2014) shows that it is at least possible to automatically generate interface endofunctors by annotating data type declarations. Future work could apply this technique to our encoding to reduce the boilerplate necessary to define coalgebras.

- Necessary type annotations. In order to apply coalgebra composition or to unfold a coalgebra often explicit type annotations have to be given. With multiple composed coalgebras and many involved interface endofunctors this quickly becomes rather verbose. Let us assume we want to unfold a coalgebra that is defined over interfaces $A\,[\_]$, $B\,[\_]$ and $C\,[\_]$, given an initial state of type S. The type annotations amount to:

  $$\mathit{unfold}[((A\ \mathbf{With_F}\ B)_{\#\mathrm{Apply}}\ \mathbf{With_F}\ C)_{\#\mathrm{Apply}}, S]\,(...)$$

  The underlying technical problem is Scala's weak type inference for type aliases and type lambdas (anonymous type functions) (Brown and Phillips, 2012):

  > [...] not a single line of the compiler was ever written with them (type lambdas) in mind – *Paul Phillips, 2012.*

  Our encoding depends on type functions and aliases to express the intersection of two functors, using $\mathbf{With_F}$. The same intersection type can also be expressed using a type lambda

  $$\mathit{unfold}[(\{\mathbf{type}\ \mathit{Apply}[+X] = A[X]\ \mathbf{with}\ B[X]\ \mathbf{with}\ C[X]\})_{\#\mathrm{Apply}}, S]\,(...)$$

which again cannot be inferred. Improving the Scala compiler's support for type lambda inference would have a large impact on the syntactic weight of our encoding.

– Threading of the state. Since our solution is based on a functional encoding mutation of the state is modeled explicitly. This is the case for the code within the coalgebraic implementation and for the user code using objects constructed from coalgebras. Every state changing method call returns a newly constructed object reflecting the updated state. As a consequence, the identity of an object is not preserved. As seen in the use case example of OpenJDK streams, passing around functional objects to create a network of collaborating objects is not only syntactically challenging with our encoding.

In the following, the encoding is discussed with respect to the requirements introduced in Section 2.1.

Extensibility at Runtime. Open coalgebras, representing mixins, are first-class values in our encoding. This allows passing them as arguments to functions and creating new mixins at runtime. Our encoding shares these advantages with possibly all encodings that are based on the representation of objects as terminal coalgebras. However, introducing composition of coalgebras we also support the dynamic composition of mixins. Dynamic composition of mixins is also the foundation to extend objects with new functionality at runtime by composing the mixin underlying the object's implementation with the mixin specifying the extension.

Transparency. The function *mix*, as introduced in Section 3.2, is the foundation for transparency of extensions in our encoding. Applying decorators opaquely shadows the actual type of the decorated object. In contrast, using *mix* and the runtime type information stored as *TypeTag* in the closure of *Fix*, our encoding allows applying the extension to the *actual runtime interface* of the object. This enables full transparency of extensions. Our extension mechanism is monotonic – only new components to the intersection type can be added. This guarantees that an object can always be safely replaced with the extended version.

Late Binding. Late binding and delegation is modeled by open coalgebras. An instance of *OpenCoAlg* is a function from a coalgebra representing the late bound self-reference to a coalgebra representing the implementation of the provided interface. The open self-reference is closed only for one level of unfolding at a time. This allows updating the self-reference at every step of unfolding, possibly respecting future extensions.

Permanency. An extension in our encoding is permanent. After applying an extension, all future layers of unfolding will be performed with the extended implementation.

Type Safety. The encoding presented in this thesis supports the incremental construction of objects where every stage of construction is a valid object and can be used as such. The dependencies to other implementations are encoded in Scala's type system and thus ensure that instances are always complete and no dynamic error such as *MethodNotFound* will be raised. The embedding of the language feature of dynamic specialization into the statically typed language Scala ensures that the encoding is type correct, assuming that Scala's type system is sound. Hence, using the encoding

is more light-weight compared to fully fledged language proposals in the sence, that no type system and corresponding soundness proofs are necessary.

A possible alternative to our encoding could be to use a mutable record of functions to represent extensible objects. However, while conceptually very simple, this alternative could not provide the necessary type safety for function calls.

Integration. One could argue that there are already exist languages that support dynamic specialization of objects, such as gbeta (Ernst, 1999). However, for the purpose of integration into existing software development environments, we believe encodings are in general a good solution. While it is great to have immediate support for a feature in a language, encodings also offer other advantages:

- Languages that are already widely used and that have a rich ecosystem might yet miss a particular feature. With encodings these languages can be retrofitted with the missing feature without requiring to adapt the infrastructure, such as compilers, typecheckers or integrated development environments. Encodings also allow prototyping language features that later may be added to a language. Since no custom compiler or preprocessor is required to develop an encoding, the "costs" or effort necessary to implement a language feature embedded as an encoding as compared to implement a full programming language is expected to be much lower. For a similar argument for embedded domain specific languages we refer the interested reader to (Hudak, 1998).

- Encodings can help us to learn about a language feature by describing the semantics of the feature in the well known semantics of the host language. In our encoding it becomes visible that there are two key ingredients for dynamic specialization: Dynamic mixin composition and dynamic update of the self-reference. While we don't claim to be the first to notice the relevance of the two concepts, both find immediate correspondence in our encoding. Having a first-class representation of mixins is an essential part of achieving both.

Our solution fulfills the requirement of reusing existing infrastructure, simply by being an encoding. Thus the above mentioned benefits apply. However, it fails to enable extensions of legacy objects that have not been defined using our framework. This and other issues are addressed in an alternate encoding we developed. It is based on boxed references, mutability and refinement of the boxed content. The alternate encoding is still under development but first experiments gave promising results. The full development of the alternate encoding is not part of this thesis and is left to future work.

5.2.1 Related Work Revisited

After having discussed the technical details of the encoding, it is worth revisiting some of the related work to highlight differences and similarities.

Both, Darwin (Kniesel, 1999) and generic wrappers (Büchi and Weck, 2000) (compare Section 2.2.4), introduce the language feature of delegation *in addition* to inheritance and thereby enable some form of dynamic specialization. While we internally also use delegation to implement object composition in Scala, the approach we take in this thesis is different. Instead of including two orthogonal features (delegation and inheritance) in a language we build on only one feature: first class mixins. This enables dynamic mixin composition and dynamic specialization. In this sense, our work is closer to the gbeta language (Ernst, 1999).

Technically, our encoding is also quite similar to the calculus of incomplete objects (Bettini et al., 2004). In the calculus of incomplete objects, late binding is implemented by passing the self-reference as additional argument to all methods. What the authors call a *generator*, a function $gen\colon Self \Rightarrow Self$, creates a new record (implementing $Self$) by partially applying all methods with *self* and thus closing the self-references. Instances of open coalgebras are also functions from the self-reference to the implementation and thus are very similar to generators. An even more striking similarity is that objects in (Bettini et al., 2004) are represented as tuples that, among others, contain the closed fixed point $fix\ (gen)$ as well as the open generator gen. Keeping a reference to gen in the tuple corresponds to storing the original coalgebra in the closure of Fix in our solution. While in our approach the current state of the object is also stored in the closure of Fix, the encoding of Bettini et al. builds on mutable state which is stored in the closure of the method implementations.

As our motivation initially came from complementing object algebras, it was not our goal to encode incomplete objects. However, the calculus of incomplete objects is also a good solution to allow incremental specialization. The semantics of incomplete objects is similar to extensible objects in our encoding. An obvious difference is that methods can be called on incomplete objects, while coalgebras (representing) mixins are not usable on their own. For future work, we consider it a worthwhile endeavour to find an immediate encoding of incomplete objects. Such an encoding might be easier to use than the presented one, since it does not necessarily be "functional" – removing the need to thread the state through every method call.

Chapter 6

Conclusions

In this thesis, we showed how to encode typesafe extensible functional objects in Scala building on a coalgebraic encoding of objects. On the way, we developed macros to automatically mix Scala objects and materialize *Functor* instances to facilitate the composition of coalgebras. We recovered much of Scala's expressiveness like self-type annotations, late binding, references to the base (similar to super) and private state building on standard techniques of functional encodings. On top, leveraging the composition of coalgebras and the first-class representation of the greatest fixed point, we were able to encode dynamic specialization of objects.

We showed, that the technique of object algebras of decomposing algorithms into traversal components can be dualized to yield dynamic specialization:

- In order to allow a modular definition of folds (that describe algorithms), object algebras take the approach to combine the advantages of functional programming and object oriented programming by encoding the style of defining functions from the first in the latter paradigm. The encoding of defining functions as *object algebras* turned a second class language feature (function definitions) into a first-class object allowing to use the modularity techniques built into the host language.
- In order to allow the modular definition of unfolds (that describe objects), the encoding presented in this thesis, *obj*.**extend**, takes the opposite approach by encoding the style of defining objects in object oriented programming using concepts of functional programming. The encoding of defining objects as *coalgebras* turned a second class language feature (class definitions) into a first-class object allowing to use the modularity techniques built into the host language.

Our approach is quite unique. Much research has been conducted to allow dynamic specialization in class-based languages. However, many of the proposals either require language extensions (Büchi and Weck, 2000; Bettini et al., 2003, 2004; Bettini and Bono, 2008) or define entirely new languages (Schmidt, 1997; Ernst, 1999; Kniesel, 1999). In contrast, we support dynamic specialization not by changing the host language but by encoding objects coalgebraically. Our encoding *obj*.**extend** can thus be distributed as a Scala library and does not require additional infrastructure other than the one usually required to build Scala programs. In particular it does not require a new compiler or custom preprocessor.

It seems that the encoding and the tools developed on the way are a good foundation for further experiments with objects encodings as became evident with the extension of selective open recursion. We hope that our encoding can help understand dynamic specialization better and at the same time that it will drive the development of new techniques for modularizing algorithms on complex data strucutures in languages like Scala.

Bibliography

M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

J. Aldrich and K. Donnelly. Selective Open Recursion: Modular Reasoning about Components and Inheritance. *SAVCBS 2004 Specification and Verification of Component-Based Systems*, page 26, 2004.

L. Bettini and V. Bono. Type Safe Dynamic Object Delegation in Class-based Languages. In *Proceedings of the Principles and Practice of Programming in Java*, pages 171–180. ACM Press, 2008.

L. Bettini, S. Capecchi, and B. Venneri. Extending Java to dynamic object behaviors. In *Proceedings of the Workshop on Object Oriented Developments*, volume 82, pages 33 – 52, 2003.

L. Bettini, V. Bono, and S. Likavec. A Core Calculus of Mixin-Based Incomplete Objects. In *Proceedings of the International Workshops on Foundations of Object-Oriented Languages*, pages 29–41, 2004.

G. Bracha and W. Cook. Mixin-Based Inheritance. *ACM Sigplan Notices*, 25(10): 303–311, Oct. 1990. ISSN 03621340. doi: 10.1145/97946.97982.

T. Brown and P. Phillips. [SI-6895] Type class instance can't be found for type lambda, but type alias works, 2012. `https://issues.scala-lang.org/browse/SI-6895`.

M. Büchi and W. Weck. Generic Wrappers. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 201–225. Springer, 2000.

E. Burmako. Scala Macros: Let Our Powers Combine!: On How Rich Syntax and Static Types Work with Metaprogramming. In *Proceedings of the Scala Workshop*, page 3. ACM, 2013.

Y. Cai. Nominal Functors. *2nd Hessian Workshop on Programming Languages*, September 2014. Talk conducted from Philipps-Universität Marburg. `http://ps-mr.github.io/hesspl-2014`.

L. Cardelli. A Semantics of Multiple Inheritance. In *Semantics of data types*, pages 51–67. Springer, 1984.

C. Clifton, G. T. Leavens, C. Chambers, and T. Millstein. MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, 2000.

W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and Its Correctness. *ACM Sigplan Notices*, 24(10):433–443, Sept. 1989.

P. Costanza, G. Kniesel, and A. B. Cremers. Lava Spracherweiterungen für Delegation in Java. In *Proceedings of the Java-Information-Days*, pages 233–242. Springer, 1999. (german language).

E. Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

M. Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, 1997.

M. Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley Longman Publishing Co., Inc., 2002.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software.* Pearson Education, 1994.

J. Gibbons and M. Johnson. Relating Algebraic and Coalgebraic Descriptions of Lenses. *Electronic Communications of the EASST*, 49 (Bidirectional Transformations 2012), 2012.

P. Hudak. Modular Domain Specific Languages and Tools. In *Proceedings of the International Conference on Software Reuse*, pages 134–142. IEEE, 1998.

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

B. Jacobs and J. Rutten. A Tutorial on (Co) Algebras and (Co) Induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.

R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

H. Kegel and F. Steimann. Systematically Refactoring Inheritance to Delegation in Java. In *Proceedings of the International Conference on Software Engineering*, pages 431–440, 2008.

G. Kniesel. Type-safe Delegation for Run-Time Component Adaptation. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 351–366. Springer, 1999.

G. Kniesel. Dynamic Object-Based Inheritance with Subtyping. *PhD thesis*, 2000.

R. Lämmel and O. Rypacek. The expression lemma. In *Proceedings of the International Conference on Mathematics of Program Construction*, LNCS. Springer-Verlag, July 2008.

O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA. ISBN 0-201-62430-3.

E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture*, pages 124–144. Springer, 1991.

B. Meyer. Applying 'Design by Contract'. *Computer*, 25(10):40–51, 1992.

L. Mikhajlov and E. Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 355–382. Springer, 1998.

T. Millstein and C. Chambers. Modular Statically Typed Multimethods. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 279–303. Springer, 1999.

O. Nierstrasz, S. Gibbs, and D. Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, 1992.

M. Odersky. Pimp my Library. October 2006. `http://www.artima.com/weblogs/viewpost.jsp?thread=179766`, Accessed: 2014-09-23.

M. Odersky and M. Zenger. Scalable Component Abstractions. *ACM Sigplan Notices*, 40(10):41–57, 2005.

B. C. Oliveira and W. R. Cook. Extensibility for the Masses. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 2–27. Springer, 2012.

B. C. Oliveira, A. Moors, and M. Odersky. Type Classes as Objects and Implicits. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, volume 45, pages 341–360, 2010.

B. C. Oliveira, T. Schrijvers, and W. R. Cook. MRI: Modular Reasoning about Interference in Incremental Programming. *Journal of Functional Programming*, 22 (06):797–852, 2012.

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

K. Ostermann and M. Mezini. Object-Oriented Composition is Tangled. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.

D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

B. C. Pierce. *Types and Programming Languages*. Massachusetts Institute of Technology, 2002.

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

T. Rendel, J. Brachthäuser, and K. Ostermann. From Object Algebras to Attribute Grammars. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages & Applications*, 2014.

R. W. Schmidt. Dynamically Extensible Objects in a Class-Based Language. In *Proceedings of the Technology of Object-Oriented Languages and Systems*, pages 294–305. IEEE, 1997.

P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse Contracts: Managing the Evolution of Reusable Assets. *ACM Sigplan Notices*, 31(10):268–285, 1996.

D. Ungar and R. B. Smith. Self: The Power of Simplicity. *ACM Sigplan Notices*, 22 (12):227–242, Dec. 1987. ISSN 0362-1340.

P. Wadler. The Expression Problem. *Java-genericity mailing list*, 1998.