Commutative Composition
a conservative approach to aspect weaving

Samuel Gélineau

A thesis submitted to McGill University
in partial fulfilment of the requirements
of the degree of Master of Science

School of Computer Science

McGill University

Montréal, Québec

December 2009

ACKNOWLEDGEMENTS

I would like to thank, first and foremost, my friend Darin Morrison, who first introduced me to Agda and with whom I have spent many hours discovering the secrets of type theory. Those who share my enthusiasm for learning these things are few, and precious.

I would also like to thank my supervisor, Jörg Kienzle, for his encouragement, his warm and continuous approval, his help, and for our exciting conversations.

I must also thank Brigitte Pientka, my other supervisor, for her numerous, pertinent, if at times harsh-sounding comments. Without her, this document might have read like a novel, but it would also have had the mathematical rigour we have come to expect from them — not a lot.

Clark Verbrugge, a professor endowed with an uncommon level of clarity, must be thanked for being the first to spot the graduate student that was hidden in me. At the opposite end of the spectrum, my employer, Environment Canada, must be thanked for keeping me grounded to the real world by giving me practical tasks. Without those reminders, the contents of this thesis might have been even more abstract and removed from reality.

Financially, I must thank my mother, my father, my grandparents, NSERC, McGill, and André Courtemanche for taking care of a good fraction of my semester fees at various moments of my graduate and undergraduate education. To those in this list

who have supported me from the beginning: I'm sorry it took so long. I'll get a job now.

My mother also had to sit through way more than her share of aspect-oriented propaganda, and I thank her for that.

ABSTRACT

Aspect-oriented programming is very good at separating concerns, but a little less at combining them back together; some aspects are simply incompatible, causing unexpected behaviours when used together. To prevent such conflicts, a conservative approach is to construct sets of aspects which are provably guaranteed to be compatible with one another. Surprisingly, to establish that two aspects are compatible, it is enough to show that they yield the same result regardless of the order in which they are woven. This principle can be used to construct, extend and transform sets of useful and compatible aspects.

ABRÉGÉ

La programmation orientée aspect a atteint son objectif, la séparation des considérations. Une fois séparées, par contre, ces considérations ne se remboîtent pas toujours parfaitement; certaines, tout simplement incompatibles, se comportent de manière surprenante lorsqu'elles sont utilisées ensemble. Afin de prévenir ces conflits, une approche conservatrice consiste à définir des ensembles d'aspects pour lesquels nous avons la preuve qu'ils sont compatibles les uns avec les autres. Pour obtenir cette preuve, étonnament, il suffit de s'assurer que ces aspects produisent les mêmes effets, qu'ils soient tissés dans un ordre ou dans l'autre. Ce principe est à la base d'une série de preuves permettant de construire, d'étendre et de transformer des ensembles d'aspects à la fois utiles et compatibles.

TABLE OF CONTENTS

CHAPTER 1
Introduction

In the last few years, a relatively new paradigm called *aspect-oriented programming* [EAK+01] has attracted enough interest for a troublesome issue to be discovered: *conflicts* [All09]. A conflict occurs when two aspects which work well in isolation exhibit an unexpected, erroneous behaviour when they are used together. While the aspect-oriented paradigm is intended to enhance modularity [Par72], and thus to allow programmers to think about each piece in isolation, the possibility of conflicts between aspects requires programmers to reason about the implementation of the other aspects, in order to determine whether a conflict will occur. The goal of this thesis is to delegate the task of preventing conflicts from the programmer to the compiler.

Our solution is to use the compiler's type checker to conservatively prevent aspects from being used together unless they are guaranteed not to conflict. The guarantee we are concerned with in this thesis is that of *commutativity*, which is true of two aspects if their joint behaviour is independent of their weaving order.

We expect that most readers will either be unfamiliar with aspects or unfamiliar with the dependent type systems in which commutativity can be expressed. We encourage those readers to continue reading, since this thesis has been written with them in mind.

1.1 Outline

In addition to presenting the outline and the contributions of this thesis, the present chapter also defines the aspect-oriented and type-theoretic terminology used in the following chapters.

In chapter 2, we list a few landmark achievements in the domain of aspect-oriented programming, as well as other domains that relate to our solution. Among these achievements is MinAML, a purely functional aspect-oriented programming language designed to be as simple as possible while retaining the core features usually associated with aspects. For our work, we strip down MinAML even further by ignoring joinpoints, pointcuts and side effects; what remains is the representation of aspects as functions, and the use of function composition to weave them together.

In chapter 3, we illustrate aspects and motivate our solution by describing examples of conflicting and non-conflicting aspects, observing that only the non-conflicting aspects commute.

In chapter 4, we show that in our stripped-down setting, commutative aspects cannot conflict, and we generalize this conclusion to other aspect representations. We also introduce the notion of an *aquarium*, a set of pairwise compatible aspects, whose elements can thus be composed without causing conflicts.

In chapter 5, we construct several example aquariums, catering for different aspect representations. We take the time to explain our aquarium-building strategy, so that readers with their own special needs can implement aquariums for the aspect representation of their choice. A major restriction is that our aquarium-building strategy only works for aspects represented as elements of an algebraic datatype.

Chapter 6 reverts to the functional representation of aspects, which, because it is not an algebraic representation, is outside the scope of the aquarium-building strategy described in the preceding chapter. In this more complicated setting, aquariums are built incrementally, one aspect at a time, by proving that each new aspect commutes with all the aspects that have already been added to the aquarium. Writing commutativity proofs is not too hard, but in order to minimize the amount of work required from the programmer, this chapter lists several short and generic proofs that the programmer is encouraged to use as lemmas.

In order to distinguish the generic proofs provided by our framework from the specific proofs required from the user of our framework, we call the former "theorems", and the latter "proof objects". Our framework is implemented using Agda [Nor09], a proof assistant and functional programming language, so the proof objects expected from the user are concrete, mechanized proofs whose validity can be verified by Agda's type checker.

We conclude our work in chapter 7 by noting that since our theory shows that commutative aspects do not conflict, since our aquarium-building strategy and our theorems allow the programmer to build collections of commutative aspects, and since the Agda type checker verifies that our theorems are used correctly, our framework is indeed letting the compiler take care of the possibility of conflicts, as desired.

Do keep in mind that in order to achieve this result, we have worked in a simplified setting where neither joinpoints, pointcuts, nor side effects have been taken into account. The last chapter describes promising research directions that could be

followed in order to improve our framework and adapt it to the imperative world with which aspect-oriented programmers are familiar.

But before attempting this challenge, let's begin by making sure that we are familiar with the topic.

1.2 Terminology

This thesis employs the tools of type theory, a very theoretical side of computer science, to tackle a problem encountered in aspect-oriented programming, a very practical side of computer science. As a result, at least half of its contents is bound to be unfamiliar to most readers.

Another problem associated with working in a solution domain disjoint from our problem domain is that terminology becomes a minefield. Many ordinary words have acquired a special meaning in one or both of these domains, making it hard to express anything in a way that readers from both domains can understand. The point of this section is to dispel those existing domain-specific meanings by anchoring our words to firm definitions.

1.2.1 Aspects

Given the subject of this thesis, we will clearly use the word *aspect* very often. It is important for us to clarify what we mean when we use the word, because even within the confines of aspect-oriented research, this word is plagued with multiple meanings.

The *aspect* in aspect-orientation, for example, is really an umbrella term for a family of techniques attempting to improve the modularity of source code beyond the offerings of object-orientation. The most well-known technique in this category

is that of the AspectJ language, which involves the concepts of *joinpoints*, *pointcuts*, and *advice*. Our approach steers away from these complications, but since AspectJ-like weaving is the most popular aspect-oriented technique, it is worth taking a few moments to explain those terms.

When a program runs, it is customary to imagine the program counter running madly along the lines of the source code. As it does so, it encounters many operations that start and finish at well-defined boundaries, such as the update of a field or the execution of a method, and those well-delimited operations are called *joinpoints*. If the program counter passes by the same line of code multiple times, it encounters distinct joinpoints, each of them an instance of the same joinpoint *shadow*.

A *pointcut* is an expression describing a subset of all joinpoints. For example, an expression denoting a particular joinpoint shadow selects all the instances of this shadow, while a method name selects all the method call joinpoints that call this method.

An *advice* is a piece of code to be inserted around a joinpoint. The behaviour of an advice is in between that of a function and that of a macro. Like a macro, the advice can decide to execute its joinpoint argument zero, one, or several times, but unlike a macro, it cannot examine its argument as code. Instead, its argument is a black box that may be called, with no arguments; a thunk, or a closure, if you like either of those terms better. In AspectJ, each advice is given access to a method called proceed which executes the captured joinpoint.

The critical difference between an advice and a function is that the programmer never calls an advice on any argument. Instead, she associates each advice with

a pointcut, and the aspect-oriented compiler inserts calls around all the joinpoints selected by the pointcut. This transformation is called *weaving* aspects into the code, and for this reason the terms *woven code* and *base code* are frequently used to distinguish between code that has been transformed in this way and code that has not yet been woven.

Finally, an *aspect* is a module which defines or associates one or more pointcuts and advices, in order to take care of a specific part of the program specification. This part is said to be the *concern* implemented by the aspect, and ideally, the programmer should be allowed to subdivide the specification into concerns in any way she pleases.

In our work, we focus on the simple case in which there is only one relevant joinpoint, and in which each aspect associates a single advice with this unique joinpoint. For this reason, we often use the terms advice and aspect interchangeably.

In our simplified setting, there are also fewer differences between aspects and functions. We ignore side effects, so the difference between executing an advice's joinpoint multiple times and using an already evaluated value multiple times can be ignored. For a large part of our thesis, in fact, we represent aspects using functions. In this, we are similar to the functional language MinAML, which, like us, composes its aspects using function composition.

One detail of the weaving transformation is very important for this thesis. Since the programmer doesn't write down the advice calls explicitly, the transformation is under-defined whenever more than one advice need to be weaved around the same

pointcut. Should aspect a be called on a piece of code consisting of a call to aspect b with the original joinpoint code as its argument, or should aspect b be the outermost call instead of a?

This choice often makes a difference in the run-time behaviour of the program, and our opinion is that the authors of existing aspect-oriented languages take this decision too lightly. To make its choice deterministic, AspectJ specifies that the aspects will be woven in the order in which each aspect lists its associations, an order to which programmers, in practice, seldom pay any attention. In addition to this default ordering, AspectJ provides a construct which allows the programmer to specify the aspect ordering she desires. But when the programmer is silent on her ordering preference, what does she intend to express? That she does not expect the order to matter? That she has not bothered to check whether the order mattered?

In any case, this minor issue of allowing the programmer to express her intention is not the reason why we discuss the case where the weaving order doesn't matter. Our primary motivation for advocating aspect compositions where the composition order is irrelevant is that those compositions are guaranteed to be free of conflicts.

1.2.2 Conflicts

A *conflict* occurs when two aspects that work well in isolation stop working when they are used together. When, on the contrary, two aspects that work well in isolation continue to work well when they are used together, the aspects are said to be *compatible*. The importance of combination and correctness in these definitions has led us to use a model of aspects where both aspects and their correctness predicates can be easily composed.

We must be careful about our use of the word *model*: this word is prone to confusion because it has acquired many different but incompatible specific meanings. Some computer scientists use the word to denote rule-based update systems such as Markov chains and Petri nets, while others use it to denote mathematical structures used to prove that a logic is consistent. In its broadest sense, however, a model is the simplification of an object of study which accounts for some features that are considered important, while ignoring others. This is why, in our study of conflicts, we use a model of aspects which ignores joinpoints, pointcuts, and side effects, to concentrate solely on the composition of aspects and on the behaviour which those compositions are expected to exhibit.

Since we are defining our terminology, now is a good time to mention the concept of *aspect interference*. Two aspects are said to interfere if one of them must modify its behaviour in order to account for the presence of the other. In this work, we do not pay a lot of attention to the distinction between a conflict and an interference; in both cases, we begin with two aspect implementations that do not work well together, but in the case of aspect interference, we subsequently discover a way to modify the implementations in order to account for the problem. Therefore, we simply view interferences as the subset of conflicts which we know how to fix.

We also want to distinguish between the expressions "to fix a conflict" and "to solve the problem of conflicts". The former refers to a specific conflict, between a particular pair or collection of aspects, which can be circumvented by changing the concerned aspects in some fashion. The latter refers to the harder problem of

conflicts in general, including those between aspects that have not yet been written nor designed. This is the problem which this thesis tackles.

Since our solution is based upon a simple model of composable aspects, however, we cannot pretend that our solution solves all conflicts in any aspect-oriented language; rather, our solution is restricted to the conflicts and languages that correspond to our model. In particular, there might be complications associated with pointcut-based weaving, non-termination, or side effects that we are unaware of, and we don't claim those to be handled by our commutative compositions.

Therefore, when we write that commutativity solves conflicts, we intend the phrase as a shorthand for the fact that under the circumstances assumed by our model, namely, in a functional programming language without side effects, pairwise commutativity in a collection of aspects is a sufficient condition for the composition of these aspects not to result in conflicts.

1.2.3 Martin-Löf Type Theory

The fact that commutativity solves conflicts is made more useful in practice by the fact that a type checker can be employed to validate this condition.

One way to make the jump from type checking to commutativity checking is to use a type system where commutativity can already be encoded as a type. Martin-Löf type theory [ML84], for example, was specifically designed to have types corresponding, through the Curry-Howard isomorphism, to propositions in first-order logic.

It is for this reason that we chose Agda, an implementation of the theory, to host our framework for writing commutative aspects. Agda is both a proof assistant and

a functional programming language, making it an ideal vehicle for both our proofs and our example aspects.

The reason Agda manages to support both proofs and programs is that in Martin-Löf type theory, the notions of *propositions* and *types* are unified into one. Because of the way our human intuition works, however, each word remains independently useful as a visualization cue and, for this reason, we often switch from one view to the other as the situation requires it.

There is a third concept, that of *sets*, that might or might not be viewed as identical to propositions and types. Martin-Löf type theory uses the word "Set" as the type of types, but this choice of terminology might lead to confusion among those who are used to sets as described by the Zermelo–Fraenkel axioms [Sup72].

The confusion arises because the axioms allow mathematicians to construct, among many other things, the union and intersection of infinitely many sets, whereas types can only be manipulated using the type constructors available in the current scope. One way to resolve the misunderstanding is to realize that in both cases, a set is an abstract construct representing a collection of elements, equipped with tools (axioms or type constructors) for forming new sets out of old ones. Classical mathematicians and type theorists simply favour different collections of tools.

In this thesis, we use the word "set" as another synonym for "type" and "proposition". For this reason, we do not assume that we can take the union nor the intersection of arbitrary sets.

Now that that we have taken care of the terminology issue, we must dispel another potential source of confusion, that of higher-order functions. To illustrate, consider the following observation.

"The fact that proofs are first-class values which libraries can create and manipulate is very handy. In this thesis, we show many proofs concerning commutativity, and each of them corresponds to a library function creating or manipulating proof objects. For example, our proof that the identity function commutes with any other function corresponds to a library function accepting a function argument, and returning a proof that id commutes with that function."

In the above text, we describe a library function concerning the identity function, and those two uses of the word "function" seem to have different meanings.

Each function, however, is simply an inhabitant of a different type. Moreover, the library function is a higher-order function, that is, a function that takes another function as its input. When we encounter these, our description will always involve functions inhabiting two different types: the type of the higher-order function, and the type of its arguments.

1.2.4 Commutativity

There is one higher-order function that has a very important role in this thesis, and that is the function composition operator. This higher-order function takes two functions f and g as arguments and returns a function which applies one function after the other. The behaviour of the resulting function usually depends on the order in which the arguments were given to the function composition operator. When this

is not the case, as expressed by the proposition $\forall x.\ f(g(x)) = g(f(x))$, the functions f and g are said to *commute*.

This "commutes with" relation should be distinguished from the arithmetic notion of commutativity. The former is a relation on unary functions, stating that the order of application does not change the result, while the latter is a predicate on binary functions, stating that the order of the arguments does not change the result. Despite the differences, the concepts are clearly related. For our purpose, one important similarity is that unary commutativity is a solution to conflicts when aspects are represented as unary functions, while binary commutativity is a solution to conflicts when aspects are represented as datatype inhabitants joined by some binary function.

In order to avoid confusion, some authors might prefer to use the word "commutability" to denote unary commutativity. In this work, however, we often want to denote the fact that both forms of commutativity can be used to prevent conflicts, and find it convenient to use the word "commutativity" as a shorthand for both solutions. Besides, the very first occurrence of the word, by François Servois in 1814 [Ser14], was used to denote unary commutativity. Under these circumstances, it would be hard to blame us for using the word with its original signification.

In any case, when two functions commute with each other, this fact is readily expressed as a mathematical proposition, and that proposition is in turn easily encoded as an Agda type. To ensure that only commutative functions are combined, then, it suffices to write an alternate combination function expecting two functions as well as a proof that the two functions commute.

In our Agda implementation, we rely on the programmer to use this alternate combinator to express that the composition order does not matter, or that she has not bothered to verify if it did. Not only does this alternative allow the programmer to better express her intention, it also encourages the programmer to double-check that intention: if a proof that the aspects commute cannot be found, it may be that the aspect ordering does matter after all.

It is important to understand, however, that our primary goal is not to help the programmer express herself better, but to solve the problem of conflicts. We have discovered that commutative composition is a conservative mechanism for avoiding conflicts, and that is our primary motivation for advocating commutativity.

1.3 Contributions

The single most important contribution this thesis has to offer is the observation that commutative aspects do not conflict.

1.3.1 The Idea that Commutative Aspects are Compatible

Here is why our central observation is not an obvious fact. Some, but not all conflicts can be resolved by changing the weaving order of the aspects involved. Such a situation is called a *precedence conflict*, and it is reasonable to expect that a solution focusing on the consequences of changing this order could detect and perhaps fix conflicts of this kind. Our commutativity criterion belongs to this class of solutions, since it requires that changing the weaving order should have no effect at all. Therefore, it is not surprising that our condition is sufficient to eliminate

precedence conflicts. The non-obvious part, however, is that our solution applies not only to precedence conflicts, but to any conflict satisfying the following constraints.

1. The aspects involved are free of side effects, i.e., they are pure functions written in a functional programming language. In particular, the aspects do not and cannot read, write, mutate, or otherwise imperatively interact with the outside world.
2. The conflict observed is expressed in terms of the value returned by the composed aspects, not in terms of properties ignored by our simplification of the problem. Commutativity does not prevent problems linked to performance or security, for example. In software engineering terms, our solution only covers deviations from the functional part of the program specification.
3. The problem observed with the composition is that the value it returns fails to satisfy a particular property which the programmer expected to be satisfied. This property must be one that was always satisfied by the values returned by one of the component aspects, and which was presumably lost in the composition because of the presence of the other aspect.
4. The property involved in the previous constraint may only examine the return value itself, not the input value. For example, if an aspect that expects a piece of formatted text always returns a new piece with the same text except that the font size has been increased, it is a fact that this aspect always satisfies the property that its output has the same text and formatting (except for size) as its input. For the purpose of describing conflicts, however, we do not consider the situation to be problematic if the composition fails to keep the formatting

unchanged, as could happen if another aspect decided to underline misspelled words, for example.

Because our central observation is both important and non-obvious, many of our other contributions stem from our desire to explain why our observation is correct.

1.3.2 Examples of Commutative and Compatible Aspects

In chapters 3 and 4, we give several examples of aspect compositions that are compatible and commutative, as well as examples that conflict and yield different results depending on their weaving order.

1.3.3 An Intuitive, Anthropomorphic Explanation

In chapter 3, we explain why the weaving order is relevant for conflicts by comparing aspects to decision makers in a company. This serves to anchor human intuition, since our wetware is better equipped for the task of predicting the behaviour of competing agents than for the task of understanding the interaction between pure algorithms.

1.3.4 A Proof that Commutative Aspects are Compatible

In chapter 4, we prove that under the assumptions listed above, commutative aspects are guaranteed to be compatible. Writing this proof is important to ensure that the intuition developed by the previous contribution is not misguided.

1.3.5 A Generalization to Alternative Aspect Representations

In the last part of chapter 4, we extend our proof to aspect representations other than functions, a generalization that can be useful in contexts where the allowable aspects need to be restricted to a known set. In this setting, the aspects

are not necessarily functions, so it does not make sense to require the aspects to be commutative, but there is an alternative requirement. If the binary function used to combine the aspects is both commutative and associative, then under a suitable generalization of the assumptions listed above, all combinations are guaranteed to be compatible.

1.3.6 Proof Implementations in a Proof Checker

Another important contribution is that we have mechanized all of our proofs using the proof assistant and functional programming language Agda. Doing so has the following important advantages.

1. Assuming the proof checker is correct, there is an increase in the confidence that our proofs are valid.
2. As it often happens, the act of mechanizing our paper proofs has led us to improve our existing arguments.
3. If the programmer wants to write a mechanized proof that her composed aspects satisfy a given property, our proofs can be used as a lemma (or as a subroutine, if the proof is viewed as a program).

Despite our enthusiasm for computer-checked proofs, we do not assume that programmers bothered by the problem of conflicts share our interest. On the contrary, our target audience consists of programmers who are happy to write individual aspects or programs without proving that these programs works as expected, but are less confident regarding compositions.

To satisfy our target programmer's imagined desire to avoid conflicts while balancing her imagined aversion for proofs against her imagined need for flexibility, we present a spectrum of conflict-avoiding contributions ranging from the simplest to use to the most flexible. Each of our contributions corresponds to a concrete collection of functions in our Agda library, which the programmer can call from her functional program (provided her program is written in Agda).

1.3.7 Simple Aquariums of Compatible Aspects

In chapter 5, we construct several aquariums, that is, several collections of aspects that are intrinsically compatible with each other. The aspects are represented by instances of common types, like booleans, natural numbers, and other algebraic datatypes. The programmer simply needs to combine those easy-to-construct values using the composition functions we provide, and the result will be free of conflicts.

In concrete terms, this contribution is simply a list of commutative and associative functions. If the aspects the programmer needs are simple enough to be expressed by single numbers, then this might be just what she needs. In practice, however, these simple aquariums are more likely to be used as primitives for building more complicated aquariums. Some of our more flexible contributions, to be described in a moment, are designed to help her in this task.

1.3.8 Unordered Pair Types for Building More Aquariums

Each example aquarium in the previous contribution is built using the following insight: it is straightforward to build a composition operator whose result does not depend on the order of its arguments if we first combine the two arguments into a form which discards this order.

Thus, in addition to the simple aquariums associated with the provided simple composition operators, chapter 5 provides unordered pair types from which custom commutative composition operators can be easily built. Allowing the programmer to write her own composition operators, and thus her own aquariums, is very useful when none of the existing aquariums are not appropriate for the task at hand.

1.3.9 A Strategy for Building More Unordered Pair Types

Types whose elements represent unordered pairs of elements of another type are not very commonly used outside of this thesis, but they are easy to construct. In chapter 5, we give a strategy for building unordered pair types based on the definition of the algebraic datatype of the elements of the pair.

We try to give the reader a good grasp of this unfamiliar concept using the many examples of the previous contribution as an illustration. At the end of the chapter, we show how to perform the transformation in general, for any datatype whose members can be well-ordered.

1.3.10 A Theorem to Ease the Transition to Functions

In chapter 6, we switch our focus back to aspects represented as functions, and present a few theorems for constructing basic aquariums employing this representation. The most useful of these theorems states that any binary function, provided that it is commutative and associative, can be curried in order to obtain a family of unary functions that commute with each other.

If this binary function is chosen to be one of the composition operators defined using one of the previous contributions, this theorem can be used as a tool to translate aquariums whose aspects are elements of an algebraic datatype into aquariums whose

aspects are endofunctions. This transition is essential to the unity of this thesis, as it shows that the unary and binary forms of commutativity are two sides of the same coin.

1.3.11 Theorems for Extending Aquariums with More Aspects

If the programmer wants more flexibility, she needs to forget her imagined dislike of proofs and write a few theorems of her own. In the remainder of chapter 6, we help her in this task by giving several theorems allowing existing aquariums to be extended with new aspects, or to be transformed into new aquariums containing a different collection of aspects.

Here are two notable theorems from this contribution.

1. Given an aquarium containing a function f, the aquarium can be extended with the function f^n, for any natural number n. The function f^n is the function which repeatedly applies, n times, the function f. If f is invertible, the theorem extends to negative exponents.
2. If M is a functor, then applying M to all the members of an aquarium transforms the aquarium into a new one, while preserving the pairwise commutativity of its elements.

In summary, we have written an Agda framework for building collections of pure functions that compose commutatively with other members of their aquarium. We interpret composition as corresponding to weaving in aspect-oriented programming, and show that under this interpretation, aspects that compose commutatively do not conflict.

CHAPTER 2
Related Work

Here are several works, both inside and outside the domain of aspect-oriented programming, that have inspired our research. Indeed, there are several other domains in which composition plays a critical role, or in which conflicts arise. Encouragingly, there is even a domain in which commutativity has been independently discovered to solve the conflicts of this domain.

2.1 Darcs

darcs [Rou05] is a version control system that shares several features with our approach. The program has the ability to compose a collection of possibly-independent components, it insists that the end result should be the same regardless of the order in which the components are added, and it guarantees the absence of conflicts when the components commute. Furthermore, since it is a version control system, it has the ability to insert (and also remove and modify) calls into a piece of code, like weaving does.

The components manipulated by darcs are patches, which darcs considers to be more fundamental than revisions. It insists that the source tree built from a collection of patches should be the same regardless of the order in which the patches are applied. Unfortunately, this is not always possible: some patches interact, depend on one another, or are simply contradictory.

It is in this domain that commutativity also has a role to play. If the patches A and B can be massaged into patches A' and B' such that the patch sequence AB yields the same result as the sequence $B'A'$, then AB is said to commute with $B'A'$. Each patch A also has an inverse patch A^{-1}.

In contrast to our model, darcs patches are not considered to be functions accepting arbitrary inputs, but bridges between two fixed revisions. For this reason, applying a patch to a revision that is not the one from which it was constructed is not possible. In particular, if A and B are both meant to be applied to a particular starting revision, then the patch sequence AB is invalid because A has moved the revision away from B's desired starting state.

Yet, when patches commute, it becomes possible to combine A and B. The trick is to consider the patch sequence $A^{-1}B$, which is valid because A^{-1} ends where A and B start. Since the sequence begins with A's inverse, this patch sequence is expected to be applied right after A. If the patches A^{-1} and B commute, we obtain a sequence $B'A'^{-1}$ that should be equivalent to the previous sequence, and which is therefore also expected to be applied right after A. In particular, B' itself expects this, and thus the patch sequence AB' must be valid. It is this sequence which is used when attempting to combine the seemingly incompatible A and B.

When A^{-1} and B do not commute, darcs complains that A and B conflict. Like us, darcs prefers the conservative approach of requiring commutativity before accepting to compose two components.

While the similarities with our domain are striking, darcs assumes that its components are invertible, which is not the case for aspects. Therefore, it wouldn't seem

like the commuting trick would be of any help to us. The fact that, on the contrary, commutativity also prevents conflicts in the absence of invertibility suggests that the link between commutativity and compatibility might be deeper than expected.

2.2 Testing for Conflicts

Commutative composition, as the title of this thesis states, is a conservative approach to aspect weaving. This means that when confronted with an undecidable problem, we have decided to play it safe, preferring to reject some correct programs rather than to accept any incorrect one. This choice entails, unfortunately, that the author of a correct program might need to work harder to get the compiler to accept her program. This is why we dedicate so much space to the theorems which might assist her in this task.

This is also why, pragmatically, the opposite choice is also appealing. By using a lightweight approach which, even though it doesn't catch all conflicts, catches enough of them, development time can be spent more usefully on adapting the code to the desires of the client, rather than those of the compiler.

Test-driven development [Bec03] is a discipline which embraces this pragmatic approach. By encouraging programmers to write automated test suites detecting the most obvious deviations from the specification, the discipline allows programmers to focus on implementing features and refactoring, without needing to worry about breaking existing code. Once we realize that introducing a new aspect also has the potential to break existing code and existing aspects, it becomes clear that test-driven development could be an asset in detecting and resolving conflicts.

[RA07] explains, however, that sometimes aspects are meant to be intrusive, to change the behaviour of a program. In this case, some unit tests are expected to fail, without this being an indicator of regression. As a solution, the paper suggests that aspects should specify, in their interface, which unit tests they expect to break. In addition, aspects can depend on (the functionality verified by) existing unit tests, or introduce their own.

2.3 AspectJ

It would be impossible to write a text about aspect-oriented programming without mentioning its most well-known language, AspectJ [KHH+01]. As a conservative extension of Java [GJSB05], AspectJ inherits its large collection of libraries, its high-end virtual machines, and its type system.

AspectJ doesn't attempt to detect conflicts, but it does provide a tool to fix some of them. Since conflicts are caused by independent aspects attempting to steer the program behaviour in different direction, one way to fix them is by asking the programmer which direction she prefers. In AspectJ, this can be accomplished using a precedence declaration, which specifies part of the weaving order.

We believe, however, that this mechanism is insufficient. Specifying which aspect should be privileged is not a solution to the problem, because the problem is not that the wrong aspect has been privileged. The problem of conflicts is that two aspects that work well in isolation may stop working when they are used together. Choosing which of the two aspects will continue to work is not a solution, since we want both aspects to continue to work.

Nevertheless, AspectJ is a great software, and without it the aspect-oriented paradigm would not have reached its current level of maturity.

2.4 CaesarJ

In addition to the pointcut and advice mechanism made popular by AspectJ, the CaesarJ language [MO03] allows classes to be combined using mixing-based inheritance [BC90]. The developers argue that each combination mechanism is appropriate in a different setting.

Inheritance makes it possible for a method to make supercalls. Given a particular inheritance hierarchy, the meaning of a supercall made by an implementation of method m in the class C is to call the m implantation in the class that precedes C in the inheritance hierarchy. Performing a supercall allows the method implementation to obtain the combined result of all parent implementations, and then to alter it in some way before returning its own version of the result. As a consequence, the very last class in the hierarchy has the final word on the end result. This situation is strikingly similar to that of function composition, a recurring topic in this thesis. When several functions are composed, it is the outermost function that has the ultimate control over the end result.

With mixins, the hierarchy order is not picked arbitrarily, but is rather the result of a specific linearization algorithm documented in the language specification. In practice, this algorithm attempts to respect the order in which the programmer has listed the mixins, thereby delegating the responsibility of picking the correct ordering to the programmer.

Our experience with the language is that we, as programmers, seldom care in which order the mixins are ordered; we see mixins as tools that add new features to our existing classes, and when we list more than one mixin, we simply want our class to be extended with all the associated features. This situation is very similar to the precedence declarations of AspectJ, in which programmers are also asked to pick an order when they just want all of the components to work.

Despite these difficulties, CaesarJ is a framework in which aspect composition plays a central role, and as such, working with CaesarJ was the spark which inspired our work.

2.5 Lisp Method Combination

Supercalls are also available in Lisp [Ste90] thanks to its advanced object system, Clos [KBdR99]. Clos supports several advanced features that are seldom available in mainstream object-oriented languages, including multiple dispatch, meta-classes, and method combination.

This last item, method combination, isn't as well known as it deserves to be. By default, methods with multiple implementations are combined in a manner analogous to supercalls: the methods are ordered in some fashion, and each implementation uses call-next-method to, as the name implies, call the next method on the list. This mechanism is the default, but the programmer is allowed to implement her own combination strategy.

The built-in method combinators are +, and, append, list, max, min, nconc, or, progn, and standard. The standard strategy has already been described, progn executes all the methods one after the other for their side effects, list runs all of the

methods and returns a list of the results, and + returns the sum of those results. The others are similar to +, but use other operations to combine the results. nconc is similar to append, but works in-place.

Among this list, +, and, or, min and max are particularly interesting because when we use them, the behaviour of the combined method does not depend on the order in which the components where linearized. We praise the combinators in this list as ideal tools to safely combine aspects, which, in this language, would be implemented as collections of partial method implementations.

While Agda is an ideal platform for writing the proofs that ensure commutativity, Lisp is an ideal platform in which to write concrete aspects that can be combined commutatively. We encourage pragmatic programmers who care less about proofs than we do to implement more commutative Lisp method combinators, and to structure libraries of composable aspects around them.

2.6 MinAML

MinAML [WZL03] is a version of the simply typed lambda calculus extended with aspect-oriented primitives. Being both functional and aspect-oriented, the language features both function applications, denoted $f\ e$, and joinpoints, denoted $l\langle e \rangle$. In both cases, the semantic is to substitute e for x in the body of the function, $f = \lambda x \to body$, or in the body of the advice, $\{l.x \to body\}$.

A critical difference between functions and advices, as also discovered independently by [Cos03], is that functions are lexically-scoped, while aspects are dynamically-scoped. Another difference, much more important for the purpose of this thesis, is

that functions have only one definition each, while advices can have several. This means that when the associated joinpoint arises, MinAML needs to combine the advices in some fashion before running the result. Like us, MinAML chose to combine its advices using function composition.

MinAML acknowledges that the order of the composition makes a difference, but unlike us, it doesn't see this as a problem. Like many aspect-oriented languages before it, MinAML simply delegates the decision to its programmers.

If the joinpoint l is bound to the advice $\{l.x \rightarrow f(x)\}$, for example, then the expression $\{l.x \rightarrow g(x)\} << e$ rebinds l to the composed advice $\{l.x \rightarrow f(g(x))\}$, while the expression $\{l.x \rightarrow g(x)\} >> e$ rebinds l to the reversed composition, $\{l.x \rightarrow g(f(x))\}$. In both cases, the evaluation proceeds with e.

We borrow from MinAML the ideas of representing aspects as functions and combining them using function composition. We ignore, however, the important notions of dynamic scoping and joinpoints, disregarding them as distractions irrelevant to the issue of conflicts. We also push composition further, by considering alternative combinators chosen by the programmer.

CHAPTER 3
Motivation

Before motivating our own contribution to aspect-oriented programming, let us motivate the aspect-oriented paradigm as it currently stands.

3.1 Aspect-Oriented Programming

The big advantage of letting the compiler weave in calls, as opposed to writing our own function calls at the appropriate locations, is that it makes it possible to neatly package code fragments that would otherwise be scattered all over a project's source tree.

In order to understand the problem of scattering, consider how we could try to solve the problem without using aspect-orientation. One of the first solutions that comes to mind is to package all of the scattered code in a single module, and to replace the original occurrences with calls into this module. This is certainly an improvement, but in fact, the new solution still has fragments scattered all over the place, because the calls themselves are part of the concern. We have merely made the pieces smaller, not less scattered.

To really modularize the pieces, it is necessary to eliminate the calls, replacing them with some external piece of code describing where the calls were. At compile time, calls can then be inserted at the described locations, restoring the original behaviour. And this is, of course, exactly what weaving accomplishes.

Let's make this more concrete by considering an example: an auto-save feature monitoring document changes, and saving those dirty documents when they are left idle for too long.

In order to do its job, the module implementing the auto-save feature needs to know when documents are modified. One way to accomplish this is to add code to all the tools in the program, making them mark the document on which they act as dirty. A better solution is to require tools to interact with documents through a narrow interface, and to mandate each method of this interface to mark the document as dirty in addition to performing its other tasks.

Using the technology of aspect-oriented programming, an even better solution can be found. The code to mark a document as dirty can be written in a separate module, an aspect, and associated with the appropriate methods of the document interface. In terms of separating concerns, this strategy is much better, since it leaves each of those methods free to perform their main task, and only their task.

3.2 Conflicts

Aspects seem wonderful because they can separate concerns that used not to be separable, but there is a dark side to them. Keeping the concerns separate makes it easier to think about each concern in isolation, but makes it harder to think about their behaviour as a whole. The reason for this is that in a system as decentralized as an aspect-oriented language, there are no global rules regulating the overall behaviour, only a set of independently enforced rules competing for power. In some sense, conflicts occur when two aspects want different outcomes.

One colourful metaphor for understanding conflicts is provided by comparing aspects to decision makers in a company. This metaphor is particularly intuitive because it relies on the behaviour of competing agents, something the human brain has evolved to predict particularly well.

Furthermore, our technical solution to conflicts — commutativity — corresponds to a neat social solution in the world of our decision makers. This should allow readers without a technical background to understand the gist of our solution to the problem of conflicts.

3.3 Aspects as Decision Makers

Imagine a company whose decisions are made by a group of decision makers, each of them concerned with a different aspect of the company's impact: its public image, its financial profitability, its environmental footprint, and so on. Each decision maker corresponds to an aspect, whose goal is to realize different parts of the program specification. For the benefit of the company, it would be best if the actions of the company reflected the opinion of all the decision makers. This corresponds to our desire for the concerns of every aspect involved to be addressed in the weaved code.

Unfortunately, the decision makers are organized into a hierarchy, where the decider at the bottom of the ladder writes a report to her superior, who considers this report as one of the many factors to consider in the report she writes to her own superior, and so on until the top echelon is reached and the final report written by the top decider embodies the final decision of the company as a whole. This hierarchy corresponds to the aspect weaving order. Indeed, the output of the first aspect applied is given as input to the second aspect, whose output is given to the

next aspect, and so on until the output of the outermost aspect is used as the fully-weaved value.

The ladder organization favours the decider who stands at the top of the hierarchy. Her opinion is guaranteed to be reflected in the final report, while the opinions of the other decision makers might be ignored or distorted. Producing such a distorted report is the situation which corresponds to a conflict. In isolation, an aspect returns a good value addressing the concern associated with the aspect, just like a lone decision maker writes a report representing her own opinion adequately. But in the presence of an outer aspect, corresponding to a superior in the hierarchy, the final value may fail to address the original concern because the outer aspect might have replaced the good value with one that breaks the concern.

Now, how might one attempt to solve the problem of distorted reports? Clearly, the hierarchy is to blame, and we need to find a way to ensure that the high status decision makers give enough importance to the low status decision makers.

3.4 Commutativity as a Criterion for Fairness

There is a famous technique for resolving cake-cutting conflicts that is relevant here. If one person is given absolute control over the division of a cake among several parties, this decider might be tempted to allocate herself an unfairly large portion of the cake. But if the person who cuts the cake gets the last choice among her selection of pieces, she will be motivated to cut the pieces as evenly as possible.

In general, to convince a decision maker to carry out her decision in a way that makes her indifferent regarding which one of several possibilities will occur, it

suffices to postulate an adversary having the avowed goal of making the situation as unfavourable as possible for the decision maker, and to give this adversary the power to choose the possibility that will occur.

Therefore, in order to convince a high status decision maker to give a fair weight to the opinion of her subordinates, we could imagine such an adversary who would reorder the hierarchy ladder at will, for the sole purpose of causing inconvenience to those who would take advantage of their position in the hierarchy. With such an adversary in place, we observe that the final decision no longer distorts the opinions of the low status decision makers, as it is no longer advantageous for the high status deciders to write distorted reports.

Of course, this comparison with decision makers is only an analogy. An important flaw in the comparison is that aspects are not competing agents, but blind algorithms doing their job in the only way they can. This means that regardless of the incentive, whether an adversary or something else, an aspect will never take the incentive into account, for the simple reason that it lacks the ability to make its own decisions.

Instead of giving incentives to the decision makers, a more mathematical approach is to restrict our attention to those who already act in the manner desired. Imagine, for example, that we want to select a company in which to invest. Further assume that all of the available companies are organized as before, as a hierarchy of decision makers who may or may not weight the opinion of their subordinates equitably. If we favour a particular company feature, say, financial profitability, then

it will be profitable to restrict our attention to those companies whose top echelon is occupied by the company's director of finance.

If, instead of favouring one particular feature, we favour balanced companies who take all the aspects of a decision into account, to which companies should we restrict our attention? Clearly, those companies with distorted final reports are not candidates. So we would like to restrict our attention to those companies who, with or without an adversary to coerce them, take the same decision regardless of the hierarchical order.

In other words, we would like to restrict our attention to companies whose decision procedure is commutative. This could be shown to be the case, for example, by an independent auditing company who would shuffle the order once in a while and report any qualitative difference in the final decision.

3.5 Examples

Now that we have seen why commutativity should prevent conflicts, let's see how commutativity correlates with the lack of conflicts, in practice, by considering a series of examples.

3.5.1 Magnification

Consider an aspect which magnifies documents, making them easier to read. This could be implemented, for example, by modifying all the calls to drawing primitives occurring within the module responsible for displaying documents, doubling the arguments given to those primitives in order to make all the shapes twice as large. Or, using aspect-oriented technology, a pointcut denoting those calls could be written, and an advice doing the doubling could be associated with the pointcut.

Does this aspect conflict with the auto-save aspect described at the beginning of this chapter? It's hard to imagine a way in which they could interact. Both are concerned with documents, but their pointcuts select different parts of the code, and there is no overlap in the data manipulated by each aspect. And indeed, when both aspects are weaved in, the documents of the weaved program are both magnified and auto-saved.

Moreover, this behaviour is the same whether the magnifier is weaved in before or after the auto-save aspect: since the pointcuts do not overlap, there is no ambiguity regarding the order in which to compose the advices, so the resulting weaved code is the same regardless of the weaving order. Thus, the two aspects are both compatible and commutative.

3.5.2 Passwords

For comparison, here is an example of aspects that do conflict. Suppose you wished to extend a password-protection system so that

1. users are prevented from using passwords less than six characters long, and
2. passwords are hashed in order to avoid storing them as plain text [WG00].

One way to implement the first concern is by using a pointcut to intercept password changes. The associated advice would check the new password length, and throw an exception if the password is too short.

The second concern can also be implemented by intercepting password changes. This time, instead of examining the password, the advice transforms it using a hashing function. The advice should also be associated with the method that validates

login attempts, so that the stored hash can be compared against the hash of the user's attempt, rather than against the raw string.

In the case where the validation aspect is applied before the hashing aspect, the behaviour of the weaved program is that short passwords raise exceptions, while long passwords are accepted and then hashed, as desired. In the case where the validation aspect is applied after the hashing aspect, however, the behaviour becomes incorrect: short passwords get hashed into long hexadecimal sequences and the validation aspect considers them to be long enough, even though the original password was too short.

One weaving order yields a correct behaviour while the reverse order yields an incorrect one, so in particular, the two resulting behaviours are different. Therefore, the aspects do not commute. This example demonstrates that when aspects are not commutative, they may behave incorrectly, but they may also behave correctly. This is exactly what one should expect from a conservative criterion: commutative compositions must be compatible, but compatible compositions need not be commutative.

3.5.3 Colours and Languages

There can also be conflicts where neither order leads to a satisfactory outcome. As a very simple example, an aspect which paints a button blue can never be compatible with an aspect which paints this same button orange, because regardless of the weaving order, the colour of the button in the weaved program cannot be both blue and orange at the same time. And since each weaving order yields a button of a different colour, the aspects do not commute.

As a more complicated variation on this theme, consider an aspect which translates the text of the button into Italian, and another aspect which translates the text into Spanish instead. The core of the problem is the same: the text of the button cannot be both Italian and Spanish at the same time. But if the aspects really accomplish their job by performing language translations, as opposed to simply replacing the original string with canned text, a different problem can arise.

Assuming that both aspects are designed to translate English text into their language of choice, the composition is destined to fail. When the first aspect performs its translation, it returns a string of text that is not valid English. This might confuse the second aspect, and yield gibberish that is not in either of the three languages.

This might be an undesirable outcome, but this situation is not a conflict per se. A conflict is only considered to occur when two aspects that work well in isolation stop working when they are used together, and in this thesis we only consider an aspect to work correctly if it always returns satisfactory results, regardless of its input. Clearly, in the case that the input text is not valid English, the second translation aspect is not returning a satisfactory result.

We can even imagine a twist to the translation scenario that would make the aspects compatible after all. Suppose that the task of a translation aspect was not to return translated text at all cost, but simply to return translated text if possible, and to return the string `"#ERROR"` otherwise. The second aspect, noticing that the input text is not valid English, would return `"#ERROR"`, and this would be a satisfactory behaviour. Thus, the two translation aspects would be compatible, since they would continue to work in a satisfactory manner in the presence of each

other. Furthermore, since both weaving orders would correctly return `"#ERROR"`, the aspects would commute.

3.5.4 More Passwords

The situation with the `"#ERROR"` strings being correct and expected is similar to the password length validation aspect we have seen earlier. Its specification requires it to fail, by throwing an exception, whenever the password entered by the user is shorter than six characters.

We can extend this first password validation aspect with a family of aspects validating other desirable password properties: that the password is not a dictionary word, that it contains both letters and digits, and so on. Aspects in this family can be combined to form stronger password restrictions, enforced by checking each desired characteristic one after the other. The order in which the characteristics are checked depends on the weaving order, but the global behaviour of the composition does not: in all cases, the password is accepted if and only if it passes every single validation test.

This family of aspects with similar goals is our first example of an aquarium, because the aspects from this family can be freely combined without causing conflicts. Furthermore, if the exceptions thrown by the different aspects upon encountering weak passwords are indistinguishable from one another, then the aspects of the aquarium commute. If, on the other hand, the exceptions differ in order to explain which criterion was violated by the password, then the aspects do not commute because a password violating more than one criterion will cause a different exception to be raised depending on the weaving order.

In that case, the situation would be another benign case of compatible yet non commutative aspects, allowed by the fact that commutativity is conservative. The opposite situation, a commutative aspect that is not compatible, cannot happen, and our next chapter is dedicated to a proof of this fact.

CHAPTER 4
Solving Conflicts

In this chapter, we refine our notion of conflicts, and use our definition to prove that types and commutativity are sufficient to solve them. We repeat the same argument three times: once in the familiar setting of functions, once with aspects represented by functions, and once in general, for aspects represented using values of an arbitrary type. All three versions of our argument have been mechanized using the Agda proof assistant.

4.1 Overview

Here is a quick statement of our three arguments. The subsequent sections will restate them in more details, adding explanations and examples.

For functions, our argument is the simple observation that the image of a composition lies inside the image of the last function it applies.

For aspects represented as functions, our argument states that if two functions commute with each other, then the image of their composition lies within the intersection of their two images. Under the right assumptions, this means that the aspects represented by the two functions are compatible with each other.

For aspects in general, our argument states that when we compose aspects using an associative and commutative composition operator, the aspects never conflict. If we restrict the domain of aspects to a small subset of functions such that any two

of them commute with each other, then on this subset the ordinary function composition operator becomes commutative, and our argument for aspects represented as functions becomes a special case of our argument for aspects in general.

This strategy of restricting the aspects to a domain guaranteed not to cause conflicts is another important notion introduced in this chapter. Choosing a target aquarium is the means through which aspect implementers can collaborate to write compatible aspects, without having to examine each other's implementation details.

Before we go through each argument in detail, we need to distinguish the conflicts we are concerned with in this thesis from the simpler conflicts that a good type system could already solve.

4.2 Typed Aspects

Just as giving types to functions is useful to detect problems at compile time, giving types to aspects makes it possible to detect some of the conflicts early on.

It is necessary to distinguish types as used in this role, where it can be said that a function or an aspect has such and such type, from the broader use of types which this thesis is advocating, namely, using a type checker to enforce commutativity in order to prevent conflicts. In the later case, we use the fact that Martin-Löf terms correspond to proofs, using the type checker to verify that our commutativity proofs are valid. In the former case, the case with which this section is concerned, types are simply used to classify the values manipulated by our aspects and functions, using the type checker to ensure that the values given to and returned by them are of the type expected by the programmer.

We are used to the idea of functions being checked this way, but for aspects, it is less obvious that types are appropriate. In this section, we review the justification for giving types to functions, and see that the justification applies equally well to aspects.

4.2.1 Typed Input

Functions are implemented by abstracting over the possible arguments a parameter could receive. A function's input type characterizes this set of possible arguments. If a value of the wrong type is given as an argument, the function is likely to fail, since its author did not prepare for this possibility. To avoid such a situation, the compiler verifies at compile time that the values passed as arguments to the function are always guaranteed to have the proper type.

Aspects are also implemented by abstracting over a set of possibilities, namely, the set of possible programs in which they could be woven. Therefore, unless aspects work with any possible host program, it makes sense to classify programs into types and to assign input types to aspects.

For example, consider a security aspect intercepting database queries in order to check that the current user is authorized to access the information. If the access is denied, the aspect may throw a security violation exception, leaving it to the surrounding code to catch the exception and inform the user of the denial in a way that is appropriate for this particular application. This solution only works when the program in which the aspect is woven is indeed prepared to catch the exception, so it makes sense to check, statically if possible, that all the database queries in the base

program are within the dynamic extent of an exception-catching block expecting security violations.

Other aspects might depend on other features of their host programs, and it doesn't matter whether or not those features can, in practice, be checked at compile time. The point is that each aspect has a set of programs for which weaving is appropriate, and in this situation the traditional solution to prevent mismatches is to use types. Even when the available static analyzes are not sufficient to detect the relevant properties of the host program, types usually remain useful as a good approximation. In most languages, for example, the input type of integer division is a pair of integers, even though pairs whose second element is zero lead to trouble.

We expect that the static analyses required to detect the kind of properties needed by aspects might be different from the static analyses currently in use for functions. It is not the goal of this thesis, however, to suggest new analyzes adapted to the new domain. We focus on the simplified setting of aspect composition at a single joinpoint, discarding the remainder of the host program as an unnecessary complication. At this level of detail, aspect composition is indistinguishable from function composition, and we are thus justified to reuse the simple analyzes performed by existing type checkers. In particular, when composing two aspects represented by the functions g and f, we compare g's input type with f's output type.

4.2.2 Typed Output

In order to help the compiler determine the type of the values passed as arguments, each function is also given an output type. This type characterizes the set of

possible return values for the function, a useful information when the argument of a function is also the return value of another function. For example, suppose that the value x is a valid input to the function f, and that the function g's input type is T. If a program contains the expression $g(f(x))$, it is important for the compiler to know whether the function f returns values of type T or not.

Similarly, programs act differently when they host an aspect, and so it makes sense to associate each aspect with an output type characterizing the set of programs with that aspect just woven in. Again, in our simplified setting, we simply use the output type of the function representing our aspect. Therefore, when combining aspects g and f to obtain the composed aspect $\lambda x.\ g(f(x))$, the type checker automatically takes care of detecting type mismatches.

The reason we bring up the subject of mismatches, however, is to point out that they are not the kind of conflicts we are interested in. Type-based conflicts occur when one aspect brings its host program into a state that a second aspect is not prepared to handle; this second aspect, predictably, fails to handle the program, causing a conflict. Type systems are very good at solving this kind of problem, so if they were the only kind of conflicts around, we would concentrate our efforts on designing such a type system. But the point is that there are also some aspects that conflict even though their types are compatible.

For example, consider an aspect which colours misspelled words in red, along with an aspect which strips away all of the colours, for printing, readability or some

other purpose. The spell-checking aspect outputs coloured text, and the colour-discarder expects coloured text as input, so type-wise the aspects seem to be compatible. In the other direction, the colour-discarder outputs black text, and the spell-checker expects black text as input, so again, the types are compatible. If we compose the aspects together, however, we either obtain a black text that does not appear spell-checked, or a spell-checked text whose colours have not been discarded. In both cases, the output is incorrect (indeed, no output could possibly satisfy both requirements), so the aspects conflict even though their types are compatible.

4.2.3 Endofunctions

Since type compatibility is not sufficient to guarantee the absence of conflicts, we need a more stringent criterion. At this point of the discussion, it is no secret that the criterion we advocate is that of commutativity, that is, that two aspects f and g should satisfy the proposition $\forall x.\ f(g(x)) = g(f(x))$. The remainder of this chapter is dedicated to explaining why this criterion is sufficient, but for now, we examine the consequences for the types of compatible aspects.

In the previous example, we had two aspects f and g (a colourizer and a decolourizer) that were type compatible, whether they were composed as $\lambda x.\ f(g(x))$ or as $\lambda x.\ g(f(x))$. Those two compositions appear in our definition of commutativity, and both expressions need to type check before we can evaluate them and compare the results for equality. For this reason, we always need the candidate aspects to be type compatible in both directions before we can consider whether the candidate aspects are commutative.

This double type requirement restricts the available types to a large extent. Let's explore this restriction more precisely by applying standard type unification techniques to the term $f(g(x)) = g(f(x))$.

We start by giving the variable x an arbitrary type X, and by giving the functions f and g the arbitrary arrow types $A_1 \to B_1$ and $A_2 \to B_2$. Then, we refine our type assignments by noticing that x is given as an argument to both f and g, so in order for the application to type check, the types A_1 and A_2 must both be equal[1] to X.

Then, since f receives a value returned by g, we learn that f's input type A_1 must be equal to g's output type B_2. Similarly, since g receives a value returned by f, we learn that type A_2 must be equal to B_1. The conclusion of this unification exercise is that all of the type variables must be the same: the variable x has type X, and both functions have type $X \to X$.

A function whose type has this form is called an *endofunction*, and we use this kind of function to represent our aspects. We use endofunctions a lot, so we use the abbreviation $X \circlearrowleft$ to denote the type $X \to X$.

Moreover, in the following sections we are concerned with commutativity only, and would like to avoid distracting the reader by further references to types. For this reason, we fix a type X, and state now that all the otherwise arbitrary functions

[1] In the presence of subtyping, X merely needs to be a subtype of both A_1 and A_2. As yet another simplification, we ignore this issue.

used in the remainder of this chapter shall be chosen so that their input and output types are X. This makes sure that the compositions we express are type compatible, and also fixes a set from which subsets can be drawn.

4.2.4 Subsets

A predicate P on values of type X is an intuitive representation for subsets of X, mimicking the set comprehension notation $\{x \in X \mid P(x)\}$.

This representation gives subsets useful properties which sets don't have. We have already mentioned that union and intersection are not defined on the sets of Martin-Löf type theory. Predicates, on the other hand, can be combined using logical combinators such as $\wedge$ and $\vee$. Using them, it is straightforward to define union and intersection for subsets. Given two predicates P and Q representing the subsets $\{x \in X \mid P(x)\}$ and $\{x \in X \mid Q(x)\}$, we can construct the union and intersection of the two subsets as $\{x \in X \mid P(x) \vee Q(x)\}$ and $\{x \in X \mid P(x) \wedge Q(x)\}$, respectively.

Armed with this new vocabulary, we are now ready for the details of our three arguments.

4.3 Unary Functions

In this section, we temporarily forget about aspects, focusing instead on pure functions of one argument. Working in this familiar setting makes our argument much easier to visualize. Afterwards, we repeat our argument using aspects, focusing on the characteristics which distinguish aspects from functions.

The unary function version of our argument states that the image of a composition lies inside the image of the last function it applies. But what does this mean? Once we define our terms precisely, the observation will appear self-evident.

4.3.1 Definitions

The *image* of a function f is the subset $I_f = \{y \in X \mid \exists x.\ f(x) = y\}$ of values that can be returned by f given the right input.

The *composition* of two functions g and f is the function $g \circ f$, or sometimes just gf, which first applies f to its input and then applies g to the result. The right-to-left order of the notation follows from the mathematical notation for function application: for any input x, the application $gf(x)$ is defined by $g(f(x))$.

Composition is associative, since $(h \circ g) \circ f$ and $h \circ (g \circ f)$ are both defined by $hgf(x) = h(g(f(x)))$. Composition is not commutative, however, since it is not true that $fg(x)$ is always equal to $gf(x)$. For example, if $f(x) = 10 + x$ and $g(x) = 2 \times x$, then $gf(100)$ gives 220 while $fg(100)$ gives 210.

We will soon be interested in pairs of functions f and g such that $fg(x) = gf(x)$, but for now, the observation we wish to make does not depend on this property.

4.3.2 Proof

We wish to show that the image of a composition $g \circ f$ lies inside the image of g. In other words, we want to show that if a value y is in the image of gf, then it is also in the image of g.

The observation is straightforward, barely deserving to be called a proof. Since y is in the image of gf, there must be an input value x such that $y = g(f(x))$.

Therefore, there is a value x' such that $y = g(x')$, namely $x' = f(x)$. This establishes that y is in the image of g. □

4.3.3 Example

In order to give examples, we must pick a more concrete domain than our arbitrary X. Let's use Int, the type of integers. We compose the function abs, which computes the absolute value of an integer, with the function $\mathsf{plus\text{-}ten}$, which adds ten to its integer argument. Note that the function add, which adds two numbers together, is not a good choice of function because it is not an endofunction: it expects a pair of integers and returns just one.

The image of abs is the subset $\{x \in \mathsf{Int} \mid x \geqslant 0\}$ of non-negative integers, while the image of $\mathsf{plus\text{-}ten}$ is the subset $\{x \in \mathsf{Int} \mid \mathsf{true}\}$ of all integers. Each of the two compositions $\mathsf{abs} \circ \mathsf{plus\text{-}ten}$ and $\mathsf{plus\text{-}ten} \circ \mathsf{abs}$ has a different image. The image $I_{\mathsf{abs} \circ \mathsf{plus\text{-}ten}}$ of the first composition is also the set of non-negative integers, which is included in itself. The reverse composition $I_{\mathsf{plus\text{-}ten} \circ \mathsf{abs}}$, on the other hand, has the image $\{x \in \mathsf{Int} \mid x \geqslant 10\}$, which is trivially included in the subset of all integers.

In general, the image of a composition gf can be approximated by the image of g. As our last example shows, however, this approximation can be very imprecise.

4.3.4 Usage: Target Properties

The approximation can be used to check a weak form of correctness: if there is a desired property that the values in g's image possess, our argument allows us to conclude that the values returned by gf always have this desired property.

For comparison, full correctness is a very strong correctness criterion. Given any particular input x, full correctness may assert that $f(x)$ is the value required by the specification, or that $f(x)$ belongs to a subset of allowed values, as the case may be. For example, the specification of a sorting function may require the implementation to return a list having the same elements as its input list, but in an increasing order.

In general, each function f may be associated with a binary predicate $F(x, y)$ which is inhabited when the specification of f allows it to map x to y. Validating that f satisfies its specification requires writing a proof that $F(x, f(x))$ is inhabited for all x.

Proving full correctness is often regarded as too time-consuming, and for this reason programmers often settle for the partial correctness that dependent types can provide[2]. Take the function tail, for example. Given a non-empty list as input, the specification for tail asks for the first element of the list to be discarded, resulting in a list with one less element than its input list. Dependent types make it possible to reify the list length part of this specification at the type level.

Dependent types refine the classification of values into types by making some property of the values visible in their type. For example, a list of length n might be represented as a value of type List n. This type-level dependency on an integer expression makes it possible to express a type like List (n + 1) → List n, which is

[2] Some systems use dependent type to support full correctness proofs, but in this section, we are more concerned about the correctness provided by the type of an expression than by the correctness provided by accompanying proofs.

perfect for a function like tail. Partial correctness is not as precise as full correctness, but its advantage lies in the fact that instead of requiring a proof, a simple type check is sufficient to validate that a function satisfies the partial specification given by its type.

In Martin-Löf type theory, dependent types abound, but the functions we are interested in have the type $X \circlearrowleft$, which is not dependent. Nevertheless, the idea is interesting: instead of checking for full correctness, it is often easier to check that the values returned by the function belong to the correct set.

Target properties take this idea further, by checking that the values returned by the function belong to the correct subset rather than the correct set. We associate each function f with a unary predicate F such that $F(y)$ is inhabited whenever the output y satisfies a desired property. This predicate is much weaker than full correctness, which uses a binary predicate, because the unary predicate cannot compare the output with the corresponding input. For example, it is possible for the unary predicate to require that the elements of the output list should appear in increasing order, but it is not possible to require that the output list should have the same elements as the input list.

Proving that a function satisfies this weaker correctness criterion might be easier than proving full correctness. In particular, if $F(y)$ is implied by $I_f(y)$, then since $I_f(f(x))$ is true for all x, we can conclude that $F(f(x))$ is also true for all x. When the latter is true, we say that f itself (as opposed to just one of its outputs) satisfies the target property F.

As an extension to this proof strategy, and still under the assumption that $F(y)$ follows from $I_f(y)$, our argument allows us to conclude that the composition fg always satisfies the target property F, regardless of our choice of g. This may or may not be useful in proving that the composition also satisfies its own target property, but as we show in the next section, for aspects it is a useful fact to know.

4.4 Aspects as Functions

Let's go through the argument again, using aspects instead of functions. Consider two aspects represented by the functions f and g, each with its own image. Since the images are subsets, not sets, we can take the intersection of the two images and obtain another subset. Our argument, this time, is that if the functions commute, then the image of the composition gf lies within this intersection. In other words, we wish to show that commutativity and $I_{gf}(y)$ entails $I_g(y) \wedge I_f(y)$.

4.4.1 Proof

This proof begins in the same way as our unary function proof. Since y is in the image of gf, there must be an input value x such that $y = g(f(x))$. Therefore, there is a value x' such that $y = g(x')$, namely $x' = f(x)$. This establishes that y is in the image of g.

Now that we have shown $I_g(y)$, we complete our proof by showing $I_f(y)$. Again, there must be an input value x such that $y = g(f(x))$. But since the functions commute, y is also equal to $f(g(x))$. This gives us a value $x'' = g(x)$ such that $y = f(x'')$, establishing that y is also in the image of f. □

4.4.2 Example

Let's consider our colourizer/decolourizer pair of conflicting aspects again. They are type compatible, but they are not commutative since one composition order yields black text while the other yields coloured text. Worse, they are not even endofunctions.

To turn them into endofunctions, define a type T of possibly coloured text. Our new colourizer f should not expect its input to be black anymore. Let's make it paint misspelled words in red while leaving correct words in their original colour, even if they are red. After all, they might be red because they were determined to be grammatically incorrect, rather than misspelled. Similarly, our new decolourizer g should be able to handle text that is already black, which is straightforward enough.

With these modifications, both f and g now have type $T \circlearrowleft$, they are endofunctions, they are type compatible; but they still conflict. It is still the case that, depending on the composition order, the result is either black text that has not been spell-checked, or spell-checked text whose colours have not been discarded.

The key phrase is *depending on the composition order*. The two aspects are clearly not commutative.

To make them commutative, define a type T' of formatted text. In addition to being possibly coloured, formatted text can be made bold, italic, or underlined. Instead of colouring misspelled words in red, our new spell-checker f' underlines them, leaving the rest of the format the same. Our new decolourizer continues to make all words black, but this time it preserves the remainder of the formatting,

making it possible to have a result that is both black and spell-checked. Moreover, since words are just as misspelled when they are black as when they are coloured, the same words end up underlined (and black) regardless of the composition order. Our new aspects commute.

Our new aspects f' and g' have type $T' \circlearrowleft$, they are endofunctions, they are type compatible, and they commute. This makes them an ideal pair to illustrate our argument. The image of f' is the subset of texts whose misspelled words are underlined, while the image of g' is the subset of texts which don't use any colour except for black. The intersection of those two subsets consists of the the black texts whose misspelled words are underlined. The image of the composition is the same as the intersection, so it trivially lies within itself.

Commutativity is an essential player in this apparent coincidence. The aspects f and g, for example, are not commutative. The image of f is the subset of texts whose misspelled words are red, and the image of g is the subset of texts whose words are all black. The intersection of those two subsets consists of the black texts whose words are all spelt correctly. The image of fg consists of the texts whose words are all black except for the misspelled ones, which are red, while the image of gf consists of all black texts. The image of neither of those two compositions lies within the intersection of I_f and I_g: without commutativity, there is no coincidence.

4.4.3 Usage: Absence of Conflicts

The image of g is still useful as a poor approximation of I_{gf}, but when g and f commute, the intersection $I_g \cap I_f$ is a better approximation.

This new approximation can be used to check another form of correctness: if there are two desired properties which are possessed, respectively, by the values in the image of g and the values in the image of f, then our argument allows us to conclude that the values returned by gf always satisfy both properties. To see why this is the case, consider that any value returned by gf lies in I_{gf}, and thus in the intersection of I_g and I_f. Because the output value lies in the image of g, it satisfies g's target property, and since the output value also lies in the image of f, it must also satisfy f's target property.

This form of correctness is precisely the kind of correctness which is appropriate to describe the absence of conflicts. Indeed, if two aspects work well in isolation, then each must satisfy its own target property. And if the composition of these two aspects satisfies both target properties, then the aspects are seen to work well when they are used together.

But why should a composition be said to work well if it satisfies the appropriate target properties? Wouldn't it be more accurate to use full correctness, which can describe the expected behaviour more precisely than a target property? The answer, as the next section shows, is that using full correctness leads to contradicting intuitions, whose resolution involves target properties.

4.4.4 Problems with Full Correctness

In our most recent spellchecking example, it is clear that the goal of the aspect is to underline all misspelled words. Other aspects could break this functionality by discarding some of the underlining or by adding new misspelled words. If the added words are underlined, however, then it is still the case that all the misspelled words

are marked, so when composing the spellchecker with an aspect that adds underlined words, the functionality of the spellchecker is not broken.

The important thing to notice is that the composition is allowed to do some changes, namely, adding underlined misspelled words, that the spellchecking aspect is not allowed to perform on its own: in order to describe the expected behaviour as precisely as possible, the spellchecker's full correctness predicate requires the text and most of its formatting to remain unchanged. Since the predicate of the composition does not require this, the full correctness predicate of the composition seems to be less strict than the spellchecker's predicate.

This observation is hard to reconcile with the intuition that an aspect composition should do everything the first aspect is supposed to do, in addition to doing everything the second aspect is supposed to do. According to this intuition, the correctness predicate of a composition should be more strict than the predicates of its component aspects.

Our solution involves dividing the correctness of the spellchecker into two requirements. First, as a specific requirement, misspelled words should be underlined. Second, as a generic requirement, the text and its formatting should be identical to the original, except where this would violate the first requirement. If we systematically split correctness predicates in this way, the specific requirement becomes sufficient to uniquely describe a correctness predicate, as the generic part is always derived from the specific part in the same way: any property not required to change by the specific requirement should be preserved by the transformation.

Notice that strengthening a specific requirement so that less values satisfy its unary predicate has the consequence of requiring implementations to alter a larger subset of the inputs in order to make them satisfy the predicate. In turn, this reduces the number of properties (if only the properties of the form "the value is equal to x") that can be preserved while satisfying the specific predicate, thereby making the generic part less strict.

Let's now revisit our example composition while keeping our new requirement format in mind. We want to allow the composition to change the text, breaking the spellchecker's generic requirement, as long as the misspelled words remain underlined, that is, as long as the change doesn't also break the spellchecker's specific requirement.

At the same time, we want the composition to perform the tasks of both of its component aspects. If we interpret this to mean that the composition should satisfy the full correctness predicates of both aspects, including the specific and generic parts of those predicates, then we arrive at a contradiction: we want to break the spellchecker's generic requirement, and at the same time we want to satisfy it. A much more reasonable correctness predicate is that the composition only needs to satisfy the specific part of the requirements of its component aspects.

We can now see why the new format reconciles our two seemingly contradicting intuitions stating that the requirements of the composition should be both more and less strict than the spellchecker's requirements. Looking more closely, it turns out that it is the specific requirement of the composition that is more strict that the

spellchecker's specific requirement, while the generic requirement of the composition, on the contrary, is less strict than the spellchecker's generic requirement.

4.4.5 Advantages of Target Properties

The previous section has solved a problem with full correctness by dividing it into two parts, the first of which (in our example at least) is a target property. Unfortunately, not all correctness predicates can be divided this way.

For example, consider an aspect whose task is to obfuscate parts of the text using the famous ROT13 cipher. The image of such a transformation, unfortunately, contains all the possible strings, so it is not possible for a target property to distinguish correctly obfuscated text from incorrectly obfuscated text.

Despite this incompleteness, we continue to use target properties as our correctness predicates of choice. Chiefly, we consider a composition to work correctly if it satisfies the target properties of both of its component aspects. This definition is directly relevant to the notion of conflicts, since they are defined to occur when two aspects that work well in isolation combine into a composition that doesn't work well anymore.

Let us reiterate our proof that commutative aspects are compatible. Since the component aspects work well in isolation, we can assume that they always return values satisfying their respective target property. In other words, we assume that if a value belongs to the image of one of the aspects, then it must satisfy that aspect's target property. But as we have shown earlier, the composition of two commutative

aspects continues to satisfy any property entailed by the image of either aspect, so the composition must satisfy both target properties, and therefore there is no conflict.

The fact that our choice of correctness definition leads to a solution to the problem of conflicts is convenient, but it is important to understand that this is not the reason why we chose it.

Among the kind of correctness predicates we are aware of, target properties are unique in that they compose well. With full correctness, for example, we have seen how asking a composition to satisfy two full correctness predicates simultaneously could contradict our expectations. With target properties, on the other hand, we can simply intersect the subsets of allowed return values in order to obtain a refined subset.

The ability to compose correctness predicates is a very important first step toward the resolution of conflicts. If there were no way to discover the behaviour expected by the programmer when a particular set of aspects are combined, then it would be impossible to guarantee that the behaviour of the composition does not deviate from this unknown expectation.

With target properties, it is possible to compute the expected behaviour of a composition. Furthermore, our proof shows that this expected behaviour is dutifully followed by the composed program. For this reason, any deviation actually observed by the programmer cannot be due to the composition, but to some other factor. For example, one of the component aspects might itself deviate from the programmer's expectation, causing the composition and its computed expected behaviour to deviate as well.

Such deviations could certainly be avoided if the programmer took the time to prove that her component aspects worked as expected, but the fact that she does not need to write such proofs in order to use our system is convenient. Our proof derives the expected behaviour of the composition from the actual behaviour of the components, so as long as the programmer is confident that her component aspects satisfy the unwritten target properties that she has in mind, she should be just as confident about the quality of their composition.

4.5 Aspects in General

Let's go through the argument one last time. In this version, instead of assuming that aspects are represented using functions, we generalize our argument to accommodate arbitrary representations. This requires a new composition operator to be used, as function composition is no longer appropriate. This version of our argument states that choosing an operator that is associative and commutative is sufficient to avoid conflicts.

Just like the previous version of our argument fixed a set X denoting the input and output domains of the function representations, we fix a set A containing all the aspects in a fixed but arbitrary representation.

We also fix an unnamed aspect composition operator which we assume to be closed over A, associative, and commutative. We continue to use the juxtaposition notation ab to denote the composition of two aspects a and b, and we also assume that there is an identity aspect, denoted 1, such that $1a = a = a1$ for any aspect a.

4.5.1 Definitions

We rely once again on the concept of target properties, which continue to be predicates on (or subsets of) the surrounding fixed set. This time, however, the fixed set represents aspects, not output values, so target properties need to be interpreted a little differently.

In order for an aspect represented as a function to satisfy its target property, we have said that the function needs to always return values satisfying that property. Now that the predicate is defined on aspects, not output values, we need a variant of the image concept which denotes a set of aspects, not a set of output values. We define the image of an aspect a as the subset $I_a = \{b \in A \mid \exists c.\ b = ac\}$ of aspects that can be obtained from a by composing other aspects with it. Since the set of aspects contains an identity element, the aspect a belongs to its own image.

As usual, we assume that if aspect a works well in isolation, this means that its target property entails the predicate I_a. For example, if aspects are represented by integers and the aspect composition operator is multiplication, then a suitable target property for the aspect represented by the integer 4 could be $\{b \in A \mid \exists c.\ b = 4 \times c\}$, requiring the end result to be a multiple of 4. This predicate coincides with I_4, and is thus trivially entailed by it.

In practice, target properties are chosen such that once aspect a is woven in, adding more aspects cannot break its target property. In those circumstances, it is hardly surprising that compositions retain the target properties or their component aspects. Still, let's make sure it is the case by writing a proof.

4.5.2 Proof

Given any pair of aspects a and a', we want to show that the image $I_{aa'}$ of the composition aa' lies within the intersection of the images I_a and $I_{a'}$.

First, pick a value b within the image $I_{aa'}$. By definition, there must be a value c such that $b = (aa')c$. Since the aspect composition is associative, we can rewrite this as $b = a(a'c)$, which establishes that b is in the image of a.

Since the aspect composition operator is also commutative, we can rewrite b as $a'(ac)$, establishing that b is also in the image of a'. □

4.5.3 Example

In order to give an example, we need to choose an aspect representation and an aspect composition operator. Let's use the type Nat of natural numbers, and use addition to compose our aspects. Clearly, zero is going to be our identity element.

This composition operator is certainly associative and commutative, but what does it mean? A number is quite an unusual representation for an aspect, and deserves an explanation.

Custom representations for aspects are useful when one wants to limit the kind of operations the aspects can perform. Our example simulates a situation where the available actions are indeed quite limited: there is only one action, say, increasing the font size of a piece of text by one point. Instead of representing this action using a function that changes the text size, we simply use the number one to denote that the text should be increased by one point, and the number zero to denote that the text size should not be changed.

When composing two such aspects, it becomes necessary to represent an aspect which increases the font size twice in a row. Compositions involving this new aspect in turn require representations for aspects which increase the font size three and four times, and in general we end up needing aspects for an arbitrarily large number of repeated applications. For this reason, we have chosen to represent our aspects using natural numbers, where the natural number n represents an aspect which increments the font size n times, or alternatively, an aspect which increases the font size by n points, once.

Regardless of the behaviour represented by aspect n, its image is the subset $\{b \in \mathsf{Nat} \mid \exists c.\ b = n + c\}$ of natural numbers that can be obtained from n by adding some amount c. This subset is just the natural numbers greater than or equal to n. Given a second aspect m, whose image is the natural numbers greater than or equal to m, we can intersect I_n and I_m to obtain the subset of natural numbers greater than or equal to both n and m. Using the max function to compute the maximum of two numbers, we can say that the intersection $I_n \cap I_m$ is equal to $I_{\mathsf{max}(n,m)}$.

If we now compare with I_{n+m}, the image of the composition, we can see that the image of the composition is indeed contained in $I_{\mathsf{max}(n,m)}$, for the simple reason that the sum $n + m$ is greater than or equal to $\mathsf{max}(n, m)$. Indeed, any value b in the image of I_{n+m} must be greater than or equal to $n + m$, and thus greater than or equal to $\mathsf{max}(n, m)$, and thus in the image $I_{\mathsf{max}(n,m)}$.

4.5.4 Usage: Absence of Conflicts

The interpretation, once again, is that the intersection $I_a \cap I_b$ is a good approximation for the image I_{ab}, and that this approximation makes it easy to check a form

of correctness involving target properties. More precisely, if there are two properties entailed, respectively, by the image of a and by the image of b, then the composition ab continues to satisfy both properties, as do all further compositions of ab with other aspects.

To see why this is the case, consider the aspect ab or any other composition obtained from it. By definition, our chosen aspect is in the image of ab, and thus in the intersection of I_a and I_b. Since our chosen aspect lies within the image of a, it must satisfy the first property, and since it also lies within the image of b, it must also satisfy the second.

What corresponds to the issue of full correctness versus target properties in this generalized setting? Quite simply, full correctness enforces a specific behaviour, and thus a specific aspect. In our font size example, the full correctness predicate of aspect n would state that the font size should be increased by exactly n points. This is too restrictive, just like it was with aspects represented as functions. If two aspects n and m are composed, their composition cannot satisfy both of their full correctness predicates at the same time, except in the trivial case where $n = m = nm$.

Target properties, on the other hand, compose well. A target property denotes a subset of aspect representations, which can easily be intersected with another subset in order to form a composed predicate. And as usual, if the aspects we are interested in do use target properties to characterize their correctness, we obtain a guarantee against conflicts between those aspects. If two aspects that work well in isolation are composed, the target property of each aspect must be entailed by the image of the aspect. This is exactly the situation which guarantees that the composition

continues to satisfy each target property, and therefore the composition also works well, and there is no conflict.

4.5.5 The Special Case of Functions

It might be hard to view this version of our argument as a generalization of the previous one, since function composition fails to satisfy our assumption that the aspect composition operator is commutative. This difficulty, however, only arises if we pick the set A of aspect representations to be the set $T \circlearrowleft$ of endofunctions with domain T. No doubt, we want to represent aspects using endofunctions; but this doesn't mean we want every single endofunction to be representing an aspect.

Our previous argument only guarantees compatibility when the aspects commute, and for this reason we should restrict our attention to a subset of endofunctions that commute with each other. Let our aspect representation type A be such a subset.

We want any pair of aspects f and g from this subset to commute with each other: for any input x, the application $f(g(x))$ should be equal to $g(f(x))$, and we carefully pick our subset A so that this equality always holds. Assuming extensionality (functions are equal when they bring equal inputs to equal outputs), this property means that for any two aspects f and g in the subset A, the function composition fg is equal to gf, so function composition restricted to A is commutative.

We also need composition to be closed over A, so we should choose our subset so that the composition of two endofunctions in A always yields another member of A. This is easy to do: given an incomplete subset A' of functions that commute with

each other, we can close the subset by defining A to be the endofunctions f for which there exists a sequence of endofunctions in A whose composition yields f. A theorem in chapter 6 ensures that if the functions in A' are pairwise commutative, so will the functions in A. Furthermore, since the sequence of endofunctions can be empty, the identity function always belongs to the extended subset. Since we are using function composition to compose our aspects, the presence of the identity function is critical, since this function is the identity 1 such that $1a = a = a1$.

Now that we have translated the setting of our argument, we must also translate our hypotheses. We want to show that a target property F satisfied by all the output values of aspect f continues to be satisfied by all the output values of the composition fg, for any aspect g in A. Since we know, by construction of A, that f commutes with g, this proposition is equivalent to the second version of our argument. In order to demonstrate this proposition using the third version of our argument, we translate F into the subset $F' = \{g \in A \mid \forall x.\ F(g(x))\}$ of aspects that satisfy F. By hypothesis, f satisfies F'. Since function composition restricted to A is both associative and commutative, we can apply the third version of our argument and conclude that the composition fg also satisfies F'. Therefore, by definition of F', all the output values of the composition fg satisfy F, as desired.

4.6 Aquariums

Each of our three arguments has been conducted, so to speak, in a clean room environment, with no undesired or mistyped aspects to distract us. In this section, we use a colourful metaphor involving aquariums to explain how to use multiple different representations for aspects within a single program.

Each aspect representation corresponds, in our colourful metaphor, to a family of *aquariums*. Each family, in turn, is divided into several individual aquariums. An aquarium is defined by choosing a specific commutative and associative operator to compose the aspects associated with the aquarium. According to the argument repeated again and again in this chapter, it is clear that aspects designed for this aquarium can be composed freely, without fear of conflicts.

In the case of functional representations, the chosen composition operator will certainly be function composition, so the task of constructing an aquarium with this kind of representation is usually reduced to finding a subset of functions on which function composition is commutative.

In order to use more than one aquarium within a single program, some *plumbing* needs to be added in order to connect the aquariums to one another. A piece of plumbing is a function that may take values out of aquariums, transform them in some way, and feed them to other aquariums. Plumbing pieces need not be endofunctions, commutative, nor associative. They are simply the non-aspect-oriented parts of the program, which aspects cannot change.

We suggest that the construction of a piece of software using aquariums should be divided into several stages, which might be performed by different groups of programmers.

In the first stage, domain-specific aquariums are designed. This stage requires some thought to be given to the kind of aspects needed in a particular problem domain, and to the kind of conflicts that may arise with those aspects. A good aquarium

design is a solution to those conflicts, working around them by making it impossible to construct the problematic situations. For example, whenever a problematic situation is found, the design could be tweaked to allow an alternative arrangement that accomplishes the same goal, that can be expressed within the aquarium, and that combine without leading to conflicts.

In the second stage, aquarium-specific aspects are designed. Programmers working on this stage can view aquariums as a platform, on which their creations may be distributed. Confident that their creations will be compatible with all the creations of all the other designers who chose the same aquarium, they are expected to explore and extend the frontier of the functionalities previously thought to be achievable within their favourite aquariums.

In the third stage, aquariums are assembled into flexible programs. Programmers working on this stage can view aquariums as tools that can increase the flexibility of their programs. By creating plumbing and attaching them to the aquariums, they can construct fully-functioning programs supporting aspect-based extensions at every aquarium. We wish to emphasize the enormous diversity of the different plumbing pieces that could be used: basically, we group all the program constructs that have nothing to do with our approach under this category. Today's programmers, writing programs without the help of our discoveries, can be seen as aquarium assemblers whose toolboxes contain no aquariums.

In the last stage, aquariums are filled with aspects. In this stage, programmers extend a specific program by acquiring existing aspects and placing them in the

various aquariums used by the program. Of course, only aspects specifically marked for the program's existing aquariums can be used.

Since the task of plumbing aquariums together into meaningful applications is so open ended, we focus on the task of constructing the aquariums themselves. The next two chapters are dedicated, respectively, to creating aquariums by writing specific commutative and associative composition operators and to creating aquariums by constructing function subsets where function composition is associative.

CHAPTER 5
Aquariums

In this chapter, we create aquariums for various aspect representations. The primary difficulty, when creating an aquarium, is to make sure that its composition function is both commutative and associative. We do not have much to say regarding associativity, which we leave for future work, so the real focus of this chapter is to explore the world of commutative functions.

Another way to state that a function is commutative is to say that its result does not depend on the order of its arguments. Written this way, it looks like the pair of arguments has a collection of properties, one of them being order, and that each property can be examined independently. If that were the case, then we could easily write a commutative function by refraining from examining this property. We could also decide to enforce commutativity, by insisting that functions should not be given the original pair, but should rather be given an altered argument giving them access to all of the original properties, except for order. Perhaps surprisingly, this unusual strategy is precisely the one which this chapter advocates.

5.1 Unordered Pairs

Let's write down our task more precisely. We have a pair (x, y) of type $\mathsf{A} \wedge \mathsf{A}$, and we want to convert it into an equivalent pair $(\mathsf{order}, \mathsf{xy})$ of type $\mathsf{Bool} \wedge \mathsf{\&A}$, for some type $\mathsf{\&A}$. Furthermore, we want the conversion to be such that the backward

pair (y, x) gets converted to $(\neg\, \mathsf{order}, \mathsf{xy})$, with this xy being the same as the one we obtained from the forward pair (x, y).

The hard part is to figure out the type $\&\mathsf{A}$ of unordered pairs. Our strategy is to start with a concrete base type A, and to cluster together the pairs of type $\mathsf{A} \wedge \mathsf{A}$ that are equivalent up to ordering. Then, we find a datatype $\&\mathsf{A}$ that has one element corresponding to each cluster.

Most of this chapter is dedicated to carrying out this strategy with several concrete choices for A. But before we do this, we need to make sure that functions written in terms of $\&\mathsf{A}$ are indeed commutative.

5.1.1 Proof

We first show that for any functions g and f whose types allow them to be composed, if the function f is commutative, then so is the composition gf.

Our proof is very simple. Given two arguments x and y, we need to show that $gf(x, y) = gf(y, x)$. But since the composition $gf(x, y)$ is defined to be equal to $g(f(x, y))$, and since f is commutative by hypothesis, we have that $g(f(x, y)) = g(f(y, x))$, as desired. □

From now on, we abbreviate such short and simple proofs as a line of equations, in which each step should be obvious enough. For example, the above proof can be abbreviated as follows.

$$gf(x, y) = g(f(x, y)) = g(f(y, x)) = gf(y, x)$$

5.1.2 Interpretation

A corollary of the above proof is that any function g of type $\&\mathsf{A} \to \mathsf{B}$ gives rise to a commutative function. To see this, imagine a function $\mathsf{convert}_A$ which converts the forward and backward pairs (x, y) and (y, x) into the values $(\mathsf{order}, \mathsf{xy})$ and $(\neg\ \mathsf{order}, \mathsf{xy})$, for some order of type Bool and some xy of type $\&\mathsf{A}$. By extracting the second component of the result, we obtain a function $\mathsf{drop\text{-}order}_A$ which brings both (x, y) and (y, x) to the same value xy. Clearly, $\mathsf{drop\text{-}order}_A$ is commutative and has type $\mathsf{A} \wedge \mathsf{A} \to \mathsf{A}$. Therefore, any function g of type $\&\mathsf{A} \to \mathsf{B}$ can be composed with $\mathsf{drop\text{-}order}_A$ to obtain a commutative function of type $\mathsf{A} \wedge \mathsf{A} \to \mathsf{B}$.

If the type B is equal to A, the resulting composition h has the type $\mathsf{A} \wedge \mathsf{A} \to \mathsf{A}$, which is just the right type for h to be used as an aspect combinator. Furthermore, h is guaranteed to be commutative, assuming $\mathsf{drop\text{-}order}_A$ is. This means that reusing the same implementation of $\mathsf{drop\text{-}order}_A$ with many different implementations for g yields many different commutative combinators. This is good news, because it reduces the burden on the programmer from one commutativity proof per combination function to one proof per datatype A (the proof that $\mathsf{drop\text{-}order}_A$ is commutative).

Furthermore, if A is an algebraic type (made using a combination of units, conjunctions and disjunctions), the burden is reduced even further since in this case our framework has the ability to generate $\mathsf{drop\text{-}order}_A$ and its proof of commutativity. We postpone the explanation of our generation strategy to the end of the chapter, preferring to guide the reader's intuition through a series of concrete examples first.

5.2 Examples

In the remainder of this chapter, we construct unordered pair type definitions for many concrete base types, along with example commutative functions written using the new unordered pair type. Because these functions are commutative by construction, we say that they are *intrinsically* commutative.

It should be noted that even though familiar functions like addition and multiplication are commutative, their familiar recursive implementations are not compatible with our insight of discarding the argument order. The problem is that in the familiar implementations, recursion is only performed on one of the arguments, leaving the other one constant. Therefore, this strategy treats the first argument very differently from the second, a fact that is incompatible with our insight that commutative implementations should not examine the order in which arguments were given. To properly ignore order, intrinsically commutative implementations must instead recur on both arguments simultaneously, stopping when they encounter a pair of arguments that can be distinguished using a criterion other than order; for example, a pair whose elements start with distinct constructors. At this point, if needed, recursion can continue on a single argument.

Our reimplementations of addition and multiplication demonstrate that adapting to the new requirement can either be very easy, as in the case of addition, or very strange, as in the case of multiplication. Our experience writing intrinsically commutative functions is that the extra constraint doesn't feel more constraining than, say, the constraint that only a constant amount of memory should be allocated to process an input of variable length.

5.2.1 Unit

Let's start with very simple datatypes, where all pairs of type $\mathsf{A} \wedge \mathsf{A}$ can easily be enumerated. The type Unit, with only one constructor, is the simplest of all.

There is only one value of type Unit, and therefore only one pair with the type $\mathsf{Unit} \wedge \mathsf{Unit}$. That pair forms a single cluster, and thus we only need one element in our type &Unit. Let's call this element u.

```
data  Unit = unit
data &Unit = u
```

According to our strategy, the commutative functions of type $\mathsf{Unit} \wedge \mathsf{Unit} \to \mathsf{B}$ are those that can be rewritten in an intrinsically commutative style, as functions of type $\&\mathsf{Unit} \to \mathsf{B}$. But since values of type $\mathsf{Unit} \wedge \mathsf{Unit}$ and values of type &Unit contain the same amount of information — none — it is straightforward to convert between one representation and the other. In turn, this means that every single function of type $\mathsf{Unit} \wedge \mathsf{Unit} \to \mathsf{B}$ can be rewritten in an intrinsically commutative style, and thus that every single function of type $\mathsf{Unit} \wedge \mathsf{Unit} \to \mathsf{B}$ is commutative.

Here is another way to see why this is true. Since values of type $\mathsf{Unit} \wedge \mathsf{Unit}$ contain no information, all functions of type $\mathsf{Unit} \wedge \mathsf{Unit} \to \mathsf{B}$ may safely ignore their arguments. In particular, they may ignore their order, and are thus indeed commutative.

5.2.2 Bool

Booleans are our first non-trivial datatype, in the sense that this time, there exist both commutative and non commutative functions of type $\mathsf{Bool} \wedge \mathsf{Bool} \to \mathsf{B}$.

With B = Bool, the functions or, and and xor are examples of the former, while implication and snd (the function which always returns its second argument) are examples of the latter. The functions or and implication are particularly interesting members of each class, because their implementations are so similar. Can we really find a criterion to distinguish them?

```
data Bool = true | false

or : Bool ∧ Bool → Bool
or (false, b) = b
or (true, _) = true

imp : Bool ∧ Bool → Bool
imp (true, b) = b
imp (false, _) = true
```

According to our strategy, the criterion to distinguish or and imp is that only or should be rewritable in an intrinsically commutative style. In order to attempt this, let us find a representation for &Bool.

Since Bool has a finite number of inhabitants, we can easily find all the pair clusters by simply listing all the pairs.

cluster	forward pair	backward pair
tt	(true, true)	(true, true)
tf	(true, false)	(false, true)
ff	(false, false)	(false, false)

```
data &Bool = tt | tf | ff
```

There are four distinct pairs of booleans, divided into three clusters: the singleton clusters tt and ff contain only one pair each, respectively (true, true) and (false, false), while the cluster tf contains the two remaining pairs, (true, false) and (false, true).

Armed with this new representation, let's attempt to rewrite our two example functions in an intrinsically commutative style.

```
&or : &Bool → Bool
&or ff = false
&or _ = true

&imp : &Bool → Bool
&imp tt = true
&imp tf = ?
&imp ff = true
```

As predicted, only the or function can be successfully rewritten. Implication is not commutative, since false implies true while true does not imply false, and for this reason imp needs the pairs (false, true) and (true, false) to map to different results (true and false, respectively). The reimplementation &imp, on the other hand, is not able to do this because it cannot distinguish between the two pairs, as they are both represented by the same value tf.

The aquarium corresponding to the or combinator can only do one thing: it can determine whether or not at least one of its aspect is true. This could be used, for example, in a system having the ability to speed up some of its functions by precomputing some of their results. If the overhead required to build the precomputation tables is significant, this ability should be used sparingly. To ensure that

the optimization is not performed needlessly, each aspect could set a flag indicating whether or not it intends to use the expensive functions often enough to warrant the precomputation, and the system would perform the optimization whenever at least one of the aspects require it.

Of course, this assumes that the aspects perform some computation in addition to setting the flag, so the behaviour of such an aspect cannot be completely captured by a single boolean. As we explain in the next chapter, however, aquariums can be combined to form aspects with orthogonal components, one of which could be a boolean flag, and the other one a computation.

5.2.3 Single

Let's try our strategy with a slightly more complicated datatype, one with three constructors. As an additional complication, one of the constructors shall contain a value of type X, for some arbitrary type X assumed to be of interest. We also assume that we already have a representation &X for its unordered pairs.

```
data  X = ...
data &X = ...

data Single = zero | one X | many
```

Note that the zero constructor takes zero arguments, the one constructor takes one argument, but the many constructor take no arguments. The reason for this is that many represents an error condition, reported when attempting to combine two aspects represented using the one constructor. The intent of the Single datatype is that only one aspect should provide a concrete value of type X; when that is not the case, the correct and expected behaviour is to report an error.

Let's figure out how to implement &Single. Since the type X is abstract, we cannot simply list all the distinct pairs this time. We can, however, list the subset of clusters whose members do not use the one constructor. This leaves us with only two constructors, a situation similar to the case of the booleans.

cluster	**forward pair**	**backward pair**
zz	(zero, zero)	(zero, zero)
zm	(zero, many)	(many, zero)
mm	(many, many)	(many, many)
⋮	⋮	⋮

data &Single = zz | zm | mm | ...

To handle the pairs containing the one constructor, we proceed by cases. First, the cluster could be of the form (zero, one x)/(one x, zero), for some value x of type X. If we want to add a constructor to &Single representing the clusters in this category, we need to parameterize this constructor with a value of type X. This makes it possible to discriminate between the different clusters that use different concrete values for x.

Here are the clusters this representation would yield if X was Bool, for example.

cluster	**forward pair**	**backward pair**
⋮	⋮	⋮
zo true	(zero, one true)	(one true, zero)
zo false	(zero, one false)	(one false, zero)
⋮	⋮	⋮

data &Single = zz | zm | mm | zo X | ...

Second, the cluster could be of the form (many, one x)/(one x, many), for some value x of type X. Again, we can add a constructor to represent this category, provided we parameterize it with a value of type X.

cluster	**forward pair**	**backward pair**
⋮	⋮	⋮
mo true	(many, one true)	(one true, many)
mo false	(many, one false)	(one false, many)
⋮	⋮	⋮

data &Single = zz | zm | mm | zo X | mo X | ...

Finally, the cluster could be of the form (one x, one y)/(one y, one x), for some values x and y of type X. How should we represent the clusters in this category? In order to help our intuition, let's list the clusters this category would contain if X was Bool.

cluster	**forward pair**	**backward pair**
⋮	⋮	⋮
oo tt	(one true, one true)	(one true, one true)
oo tf	(one true, one false)	(one false, one true)
oo ff	(one false, one false)	(one false, one false)

data &Single = zz | zm | mm | zo X | mo X | oo &X

In the case X = Bool, at least, there is one cluster in this category per element of &X. To see why this is the case in general, consider the cluster (one x, one y)/(one y, one x). Since x and y could be different, is seems that we should parameterize oo with a pair of values of type X. But if we use an ordered pair type $X \wedge X$, we end up with one value one (x, y) representing the cluster (one x, one y)/(one y, one x), and another value one (y, x) representing the cluster (one y, one x)/(one x, one y). This is not appropriate, because the two clusters have the same members and should thus have the same representation. The unordered pair type &X does not suffer from this problem, while accommodating all the different x and y that could occur.

Now that we have a definitions for &Single, let's try to write an intrinsically commutative combinator for aspects represented as values of type Single. The intent is that only a single aspect in the composition should use the one constructor, while the other aspects should use zero. As soon as two or more aspects attempt to use one in the same combination, the combinator complains by returning many. An aspect could also decide to ruin the combination by using many itself, thereby discarding any value of type X provided by the other aspects, even if there is exactly one.

```
pick-one : &Single → Single
pick-one   zz    = zero
pick-one (zo x)  = one x
pick-one   _     = many
```

The pick-one function is useful as a default combinator, for the case where it is not yet clear which combinator is appropriate for a given aquarium. Like a parameter, an aquarium is a point of variability, a point where the aquarium assembler thought

that there could be more than one acceptable way to do something. Aquariums push the idea of parameters further, by allowing more than one value to be used simultaneously at the variability point; the pick-one function simply represents the degenerate case where the semantic of using more than one value is to return an error.

5.2.4 Nat

Now that we have seen an example where one constructor had a parameter of type X, let's try a slightly more complicated case where the parameter turns the type into a recursive datatype.

```
data Nat = zero | suc Nat
```

A natural number is either zero, or the successor of a natural number. The number three, for example, is the successor of the successor of the successor of zero.

Because there is an infinite quantity of natural numbers, it is still not possible to list all the individual pairs in order to cluster them. Once again, we proceed by cases: the pairs of the cluster may either contain zero, one, or two instances of the suc constructor. Each case is similar to a case we have seen in our previous example, so rather than going through the arguments a second time, we summarize each case in the table below.

cluster	forward pair	backward pair
zz	(zero, zero)	(zero, zero)
zs x	(zero, suc x)	(suc x, zero)
ss xy	(suc x, suc y)	(suc y, suc x)

```
data &Nat = zz | zs Nat | ss &Nat
```

According to our strategy, it should now be possible to implement well-known commutative operations in an intrinsically commutative style. Let's start with the addition of two natural numbers.

```
  -- traditional implementation
add : Nat ∧ Nat → Nat
add (zero, y) = y
add (suc x, y) = suc (add (x, y))

  -- intrinsically commutative implementation
&add : &Nat → Nat
&add zz      = zero
&add (zs x)  = suc x
&add (ss xy) = suc (suc (&add xy))
```

While both the traditional and the intrinsically commutative implementations of addition proceed by recursion, there is an important difference between the two implementations. The traditional one implement addition by recursion over one of its arguments, while the intrinsic one is forced to recur over both arguments simultaneously, for the simple reason that it cannot separate them. Yet despite this difference, the two implementations are still fairly similar.

This is not always the case. Consider the following intrinsically commutative implementation of multiplication, for example, an atypical case where a clever transformation is required. For readability, we write addition using the infix (+) whenever its commutativity is not required.

```
  -- traditional implementation
mul : Nat ∧ Nat → Nat
mul (zero, y) = zero
mul (suc x, y) = y + mul (x, y)

  -- intrinsically commutative implementation
&mul : &Nat → Nat
&mul zz       = zero
&mul (zs x)   = zero
&mul (ss xy)  = &mul-suc xy where
  &mul-suc : &Nat → Nat
  &mul-suc zz       = suc zero        -- (1+0)(1+0) = 1
  &mul-suc (zs x)   = suc x           -- (1+0)(1+x) = 1+x
  &mul-suc (ss xy)  = &mul-suc xy     -- (2+x)(2+y) = see below
                    + 3 + &add xy
```

This code deserves a detailed explanation.

Multiplying by zero is straightforward, as the result is always zero. Multiplying two successors together, however, is complicated by the fact that the two numbers x and y, whose successors we are trying to multiply, are stuck together in the unordered pair xy. &mul passes the unordered pair to its helper function &mul-suc, in the hope that the helper will know what to do with it.

The task of &mul-suc is to compute the expression $(1 + x')(1 + y')$, with the added complication that neither x' nor y' can be examined directly. Nevertheless, once it is known that either of those two numbers is zero, the result is simple to compute. The tricky case, once again, is when the two numbers are known to be successors.

At this point, we need a clever trick. Since we know that x' and y' are both successors, we can write them as $x' = 1 + x$ and $y' = 1 + y$. Our task of computing

$(1 + x')(1 + y')$ is then reduced to computing $(2 + x)(2 + y)$. The clever part is that using a few algebraic manipulations, we can transform this new expression into form computable by a recursive call to &mul-suc, plus some leftover terms.

$$\begin{aligned}
(2 + \mathsf{x})\ (2 + \mathsf{y}) &= 4 + 2\,\mathsf{x} + 2\,\mathsf{y} + \mathsf{xy} \\
&= 1 + \ \mathsf{x}\ + \ \mathsf{y}\ + \mathsf{xy} + 3 + \mathsf{x} + \mathsf{y} \\
&= \ \ (1 + \mathsf{x})\ (1 + \mathsf{y})\ \ + 3 + \mathsf{x} + \mathsf{y} \\
&= \ \ \ \&\mathsf{mul\text{-}suc}\ \mathsf{xy}\ \ \ + 3 + \&\mathsf{add}\ \mathsf{xy}
\end{aligned}$$

Now that we have seen how to implement addition and multiplication in an intrinsically commutative style, let's examine the aquarium to which they give rise.

An aquarium whose combinator is addition has already been given in chapter 3: each number x can stand for an aspect which applies a specific transformation x times consecutively. Increasing the font size by one point x times and then increasing it y more times is equivalent to increasing the font size by $x + y$ points.

It is the plumbing surrounding the aquarium which determines how to translate number representations into a concrete behaviour, so it is the aquarium assembler who is responsible for choosing the action to be repeated x times. Choosing an action based on the result of other aquariums, for example, could lead to interesting constructions.

Using multiplication as a combinator could be useful in a situation where each number x represents the factor by which the represented aspect should scale a document. Scaling a document by a factor of x and then scaling it by a factor of y is equivalent to scaling it by a factor of $x \times y$.

5.2.5 Disjunctions

In addition to the type X we have assumed earlier, let's now assume a second type U, along with the type of its unordered pairs. Can we tackle a datatype having both a constructor with a parameter of type X and a constructor with a parameter of type U?

```
data  U = ...
data &U = ...

data Either = left X | right U
```

Either is the type of disjunctions, since a value of type Either could either be a thinly wrapped value of type X, *or* a thinly wrapped value of type U.

To construct its type of unordered pairs, we again proceed by cases. This time, there is one case we have not seen before: clusters of the form (left x, right u)/(right u, left x), for some x of type X and some u of type U. Clearly, we must parameterize the corresponding by a pair of values, but should it be an ordered pair or an unordered pair? Let's help our intuition by listing the clusters in the case X = U = Bool.

cluster	forward pair	backward pair
lr true true	(left true, right true)	(left true, right true)
lr true false	(left true, right false)	(right false, left true)
lr false true	(left false, right true)	(right true, left false)
lr false false	(left false, right false)	(right false, left false)

As you can see in the table above, lr true false and lr false true give rise to distinct clusters, so we do not need to combine those two cases using &Bool. This is

fortunate, because the types of x and u are different, and we have not seen how to construct heterogeneous unordered pairs yet.

In any case, we are now ready to construct &Either.

cluster	forward pair	backward pair
ll xy	(left x, left y)	(left y, left x)
lr x u	(left x, right u)	(right u, left x)
rr uv	(right u, right v)	(right v, right u)

```
data &Either = ll &X | lr X Y | rr &U
```

The Either representation for aspects is useful when there are two kinds of aspects, and one category is supposed to have priority over the other. Assuming we already have functions $\mathsf{combine}_X$ and $\mathsf{combine}_U$ to combine aspects of either priority, here is an combinator favouring the high priority left aspects. Whenever there are both left and right aspects in a combination, this combinator discards the low priority aspects, so that only the high priority aspects are left.

```
combine_X : &X → X
combine_X = ...
combine_U : &U → U
combine_U = ...

leans-left : &Either → Either
leans-left (ll xy) = combine_X xy
leans-left (lr x u) = x
leans-left (rr uv) = combine_U uv
```

An aquarium which discards low priority aspects can be useful as a generalization of the or aquarium described earlier. In the or aquarium, there are only two choices,

true or false, and as soon as a true aspect is found, all the false ones are discarded. The leans-left aquarium makes it possible to make the choices more detailed.

For example, an aspect that used to say false, to denote that it did not plan to use the expensive function often enough, could now quantify the number of calls it plans to make. Using this new information, the estimates could be combined in order to decide if, even though no individual aspect uses the expensive function often enough to warrant the optimization, sufficiently many aspects use the function to warrant it.

Inversely, an aspect that used to say true, to denote that it did plan to make a significant use of the expensive function, could now specify a range of values on which it plans to call the function. This way, the relevant ranges could be combined in order to specialize the precomputation tables on the range that will actually be used.

5.2.6 Conjunctions

In the previous section, we have mentioned the possibility of heterogeneous unordered pairs, that is, of clusters of the form (x, u)/(u, x) where x and u are of different types. How could we represent these clusters? Let's help our intuition by listing the clusters in the case where X is Bool and U is Either Unit Unit.

cluster	**forward pair**	**backward pair**
pair true (left unit)	(true, left unit)	(left unit, true)
pair true (right unit)	(true, right unit)	(right unit, true)
pair false (left unit)	(false, left unit)	(left unit, false)
pair false (right unit)	(false, right unit)	(right unit, false)

data Pair = pair X U

What has happened? We were looking for clusters of unordered pair, but we obtained the type of ordered pairs instead!

The explanation is that when we ignore the order of an heterogeneous pair, we forget in which order the arguments were given, but we do not forget which argument had the type X and which argument had the type U. Therefore, we can easily reconstruct an ordered pair (x, u) using the information we have left, even though we cannot remember whether the original ordered pair was (x, u) or (u, x).

In any case, Pair is the type of conjunctions, since a value of type Pair consists of both a value of type X *and* a value of type U.

As usual, we wish to define the type of its unordered pairs. There is only one case, the cluster (pair x u, pair y v)/(pair y v, pair x u), but this is a case we have not seen before. Surprisingly, this case is very different from the others we have seen.

One might think that the unordered pair representation might be something like **data** &Pair = pp &X &U. With this definition, however, the following two clusters (using X = U = Bool) would be mistakenly assigned the same representation.

cluster	**forward pair**	**backward pair**
pp tf tf	(pair true false, pair false true)	(pair false true, pair true false)
pp tf tf	(pair true true, pair false false)	(pair false false, pair true true)

Earlier, we pointed out that when we forget the order of an heterogeneous pair, we continue to remember which value had type X, and which element had type Y.

This is analogous to our decision, in the previous section, to represent the cluster (left x, right u)/(right u, left x) using the value lr x y: when we forget the order in which two constructs with distinct constructors were listed, we continue to remember which value was associated with the left constructor, and which value was was associated with the right constructor.

The case we are encountering now is similar. When we forget the order in which two constructs with distinct first elements were listed, we continue to remember which second element was associated with true, and which second element was associated with false. When we forget the order in which two constructors with identical first elements were listed, however, we no longer have a handle to distinguish the second elements from each other.

Since we seem to understand the case of pairs whose first element is a boolean more thoroughly than the case of pairs in general, let's define a type for this special case and write down its unordered pair type.

cluster	**forward pair**	**backward pair**
eq true uv	(tag true u, tag true v)	(tag true v, tag true u)
tf u v	(tag true u, tag false v)	(tag false v, tag true u)
eq false uv	(tag false u, tag false v)	(tag false v, tag false u)

```
data   Tagged  =  tag Bool U
data &Tagged  =  tf U U  |  eq Bool &U
```

Our representation distinguishes clusters whose pairs have different first elements, and uses the value tf u v to store the second elements u and v found, respectively, in the pair whose first element is true and in the pair whose first element is

false. In the case where the first elements of the pairs are equal, we store which value this first element is, and since we have no handle to distinguish the second elements, we store them as an unordered pair.

We will tackle the more general type Pair in the next section. For now, let us use our new type &Tagged to write an intrinsically commutative function which could be used as a combinator.

```
leans-true : &Tagged → Tagged
leans-true (eq b  uv) = pair   b   (combine_U uv)
leans-true (tf    u v) = pair true         u
```

This combinator is similar to leans-left, in that it combines aspects having two different priority levels, but this version favours the aspects that have been tagged with the priority true instead of those that have been tagged with the constructor left.

This allows aspects to tag themselves as being more important than others, if they are presumptuous enough to think so. The combinator uses $\mathsf{combine}_U$ to resolve situations where many aspects think of themselves as being highly important.

This new take on an old aquarium hints that the more general Pair type might be useful when more than two priority levels are needed.

5.2.7 Totally Ordered Types

In the previous section, we have seen how the type &Tagged needs a different constructor for the case where the booleans are identical and the case where the booleans are different. This is a good distinction to make in general, since clusters whose pairs contain identical values are very different from clusters whose pairs

contain different values. One important difference is that the latter clusters contains two distinct pairs, while the former clusters contain only one. In the light of this new distinction, let us redefine one of our unordered pair types, &X.

```
data &X = eq X | neq ?
```

Representing the clusters that have only one value is straightforward: we can simply represent the cluster (x, x)/(x, x) using the value x. It is less clear, however, how to represent an unordered pair of distinct values. Has our new distinction made our problem simpler, or more complicated?

For the moment, let's focus on ordered pairs of distinct values. Picture a table listing all the pairs of type $\mathsf{X} \wedge \mathsf{X}$, arranged in a grid. For the purpose of illustration, let the inhabitants of X be x_1, x_2, ..., x_n.

	x_1	x_2	$\cdots$	x_n
x_1	(x_1, x_1)	(x_1, x_2)	$\cdots$	(x_1, x_n)
x_2	(x_2, x_1)	(x_2, x_2)	$\cdots$	(x_2, x_n)
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
x_n	(x_n, x_1)	(x_n, x_2)	$\cdots$	(x_n, x_n)

Clearly, the elements of the diagonal are the pairs whose elements are identical, while the elements off the diagonal are the pairs whose elements are distinct. One important information revealed by this diagram is that the distinct pairs are divided into two groups of equal size, corresponding to the triangle above the diagonal and the triangle below it.

Furthermore, each cluster of distinct pairs has one pair in the upper triangle and one pair in the lower triangle. This means that we could represent those clusters using the elements of one of the triangles, say, the one above the diagonal. Is there something which characterizes this subset?

In our illustration, the elements of X are numbered from 1 to n, and this gives us a simple way to characterize each of the three regions of the table. A pair (x_i, x_j) from the upper triangle has the property that $i < j$, the pairs on the diagonal have the property that $i = j$, and those from the lower triangle have the property that $i > j$. If we assume that X can be totally ordered, we can perform this classification using the elements of X instead of using our imaginary indices.

```
data X∧X = eq X | lt (x y : X) (x < y) | gt (x y : X) (y < x)
data &X  = eq X | lt (x y : X) (x < y)
```

The above piece of pseudo-code mixes Haskell's succinct syntax for defining algebraic datatypes with Agda's syntax for dependent types. The expression lt (x y : X) (x < y), for example, is meant to define a constructor lt taking three arguments: two values x and y of type X, and one proof object of type x < y demonstrating that x is strictly less than y in the total order.

Now that we have refined the type &X, we can write a definition for &Pair, as promised in the previous section.

cluster	**forward pair**	**backward pair**
eq x uv	(pair x u, pair x v)	(pair x v, pair x u)
lt x y _ u v	(pair x u, pair y v)	(pair y v, pair x u)

```
data &Pair = eq X &U | lt (x y : X) (x < y) (u v : U)
```

This definition distinguishes between clusters whose pairs have identical first elements and clusters whose pairs have different first elements. In the first case, we store one copy of the repeated first element, along with an unordered pair containing both of the second elements. In the second case, we store the smallest first element x, along with the largest first element y and a proof that x is strictly smaller than y. After this, we store the second elements u and v corresponding, respectively, to the first elements x and y.

Now that we have a concrete definition for &Pair, we can finally implement the aquarium we have hinted at in the previous section, the one with more than two priority levels. We use values of type X to represent those levels, and we favour values that are smaller in the total order, considering them to have higher priority.

```
leans-least : &Pair → Pair
leans-least (eq x     uv) = pair x (combine_U uv)
leans-least (lt x y _ u v) = pair x u
```

5.2.8 Algebraic Datatypes

While there is a vast number of datatypes which can be totally ordered, proving that a given datatype can be totally ordered is not necessarily simpler than proving that its corresponding drop-order$_A$ implementation is commutative. For this reason, we prefer to focus on algebraic types, for which proofs can easily be generated.

Given an algebraic datatype A, our framework can generate a proof that A can be totally ordered using lexicographical ordering. This allows us, for any two values

x and y having the same algebraic type, to meaningfully define the type (x < y) of proofs that x is strictly smaller than y. This, in turn, allows us to define &A in much the same way that &X is built in the previous section. Here is, for example, an alternative formulation for the type of unordered pairs of natural numbers.

```
data &Nat' = eq Nat | lt (x y : Nat) (x < y)
```

Even though the definition of &Nat' is very different from that of &Nat, those two types contain precisely the same number of inhabitants, that is, one per distinct unordered pair of natural numbers. The actual definition generated by our framework is different still, since we use infinite, lazily generated trees to represent recursive types like Nat. In order to keep the picture simple, we prefer to skip the details, as they are not essential to understand our approach.

5.3 Limitations of the Approach

For algebraic and totally comparable datatypes, at least, the picture painted so far is rather pleasing. There is a canonical way to construct a datatype representing unordered pairs, and from this type, intrinsically commutative composition functions can be written easily.

As soon as we attempt to include non-comparable types, however, determining how to represent unordered pairs becomes a challenge. Function types, in particular, do not play well with our strategy.

In the next chapter, we examine a completely different strategy for creating aquariums whose aspects are represented as functions.

CHAPTER 6
Theorems

In this chapter, we list several simple theorems allowing function-based aquariums to be created, extended, and transformed.

The contribution of this chapter does not lie in the proof themselves; the proofs are short, obvious, and in many cases the results could have been obtained for free using the approach described in [Wad89]. Rather, we present those theorems in order to give the reader a feeling for the kind of transformations that can be done on aquariums in order to obtain more aquariums.

The intent is to show that aquarium designs are not meant to be static rules, dictating which aspects can or cannot be used in a project if the stakeholders care about safe aspect composition. On the contrary, aquariums can be extended, combined, and transformed as the need arises.

It should also be noted that the theorems listed in this chapter are not meant to be exhaustive. If needed, aquarium designers are expected to complement this collection with custom proofs of their own. Each such custom proof will benefit from our theorems, which multiply the usefulness of each proof.

For example, an aquarium implementor might need to prove that on integers (which, as opposed to natural numbers, can be negative), `++` and `--` (increment and decrement) are commutative operations. Since one of our theorems states that once

a function f has been added to an aquarium, its repeated application f^n can be added as well, we can generalize the custom proof to the much broader statement that `+=` n and `-=` m (add n and subtract m) also commute, for any choice of n and m.

To achieve this level of integration with custom proofs yet to be written, none of our proofs assume any extra structure from the aquariums they start with. The only condition for a set or a subset to be considered an aquarium is that each pair of functions f and g in the aquarium should commute, that is, for any input x it should be the case that $fg(x) = gf(x)$. Since our proofs often work at the level of function composition, it is convenient to abbreviate this as $fg = gf$, using the extensional definition of equality.

Without further ado, let's start our list of theorems by constructing a few primitive aquariums.

6.1 Starting Simple: Primitive Aquariums

The empty set, $\{\}$, vacuously satisfies the condition for being an aquarium.

From the singleton set, $\{f\}$, we can only draw one pair of functions, (f, f). Since $ff = ff$, this pair commutes, so all singleton sets are aquariums.

6.2 Extending Aquariums: Exponentiation and More

Using the exponentiation notation, we use the expression f^2 to denote the function ff. Since function composition is associative, f^2 must commute with f.

$$(f^2)f = (ff)f = f(ff) = f(f^2)$$

More generally, for any function m commuting with f, that function must also commute with f^2. Therefore, if f is part of an aquarium, it must commute with all the elements of this aquarium. Thus, f^2 also commutes with all of these elements, and the aquarium can be extended to contain f^2 as well.

$$(f^2)m = ffm = fmf = mff = m(f^2)$$

Even more generally, if two function f and g commute with all the members of an aquarium (here m is chosen as a typical member), then so does their composition.

$$(fg)m = fmg = m(fg)$$

Note that this works even if f and g themselves do not commute. In that case, f and g cannot both be added to the aquarium, so the programmer will need to choose whether she prefers adding f, g, or fg.

One case in which f and g do commute is the case in which f and g were both picked among the existing members of the aquarium. This case is very important for aspect composition, since it allows us to consider aquariums to be closed under composition. Here is what this means in practice.

Suppose we have an aspect f associated with some aquarium M. Next, we compose f with some aspect drawn from this aquarium, say, aspect g. We now have a new aspect, fg, to which we might want to add yet another aspect. But which aspects are available now? Thanks to the proof case under consideration, fg is guaranteed to commute with all of the aspects in M, therefore, the aquarium of aspects with which it is safe to compose fg is still M. This makes it much easier

to reason about aspect composition than if the set of available aspects changed each time a new aspect was added.

Back to exponentiation, we can prove by induction that if f commutes with m, then for any positive integer n, the function f^n also commutes with m. The base case $(f^1)f = f(f^1)$, where $f^1 = f$, is known to be true because of the singleton aquarium proof we saw earlier. The inductive step is as follows.

$$(f^{n+1})m = f^n fm = f^n mf = mf^n f = m(f^{n+1})$$

The above proof means that if we have an aquarium containing the function f, we can extend it with the functions f^1, f^2, f^3, and so on. Going one step further, we can also add f^0 if we define f^0 to be the identity function id. The identity function commutes with everything, since $f(\mathsf{id}(x)) = f(x) = \mathsf{id}(f(x))$.

If f has an inverse f^{-1}, we can go even further. The compositions $f^{-1}f$ and ff^{-1} are both equivalent to the identity function, and can thus be added or removed at will, without changing the behaviour of an expression. This fact is used below to show that f^{-1} commutes with all the functions m with which f commutes.

$$(f^{-1})m = f^{-1}mff^{-1} = f^{-1}fmf^{-1} = m(f^{-1})$$

Another induction proof allows us to add f^{-2}, f^{-3}, and so on.

$$(f^{n-1})m = f^n f^{-1}m = f^n mf^{-1} = mf^n f^{-1} = m(f^{n-1})$$

6.3 Converting Combinators into Aquariums

Instead of starting with an empty or singleton set, it is also possible to start with one of the aquariums we have built in the previous chapter, one based on an intrinsically commutative combination function f, and to convert it into an aquarium $[\![X]\!]_f$ whose aspects are unary functions.

To perform this conversion, suppose f is a commutative and associative function of type $X \to X \to X$. Then, for any value x of type X, the partial application $f(x)$ has type $X \circlearrowleft$, and can thus potentially be used as an aspect. For notational convenience, when we want to compose such partial applications without the parentheses getting in the way, we use the expression $[x]_f$ to denote $f(x)$. Other times, to emphasize that f is associative, we use the infix notation $x \ \langle f \rangle \ y$ to denote the full application of f to the two arguments x and y.

The aquarium $[\![X]\!]_f$ corresponding to the combination function f is the subset $I_f = \{m \in X \circlearrowleft \mid \exists x.\, m = f(x)\}$ of all such partial applications. This subset qualifies an aquarium, since the commutativity and associativity of f guarantees that any two elements $[x]_f$ and $[y]_f$ commute. Indeed, for any value z of type X:

$$[x]_f[y]_f(z) = x \ \langle f \rangle \ y \ \langle f \rangle \ z = y \ \langle f \rangle \ x \ \langle f \rangle \ z = [y]_f[x]_f(z)$$

By abuse of notation, we sometimes use an intrinsically commutative function as an index, as in $[\![\mathsf{Single}]\!]_{\mathsf{pick\text{-}one}}$ for example. This is not quite right, since $\mathsf{pick\text{-}one}$ would need to be composed with $\mathsf{drop\text{-}order}_{\mathsf{Single}}$ and then curried in order to obtain a function having the expected type $\mathsf{Single} \to \mathsf{Single} \to \mathsf{Single}$. Hopefully, the intent should be clear.

Another example is $[\![\mathsf{Nat}]\!]_{\mathsf{add}}$, or $[\![\mathsf{Nat}]\!]_+$ for short. The elements of this aquarium are annotated numbers, like $[1]_+$ and $[3]_+$. Those numbers are in fact functions, which can be composed using function composition, but this composition has the effect of adding the annotated numbers: for example, $[1]_+[3]_+ = [4]_+$. Basically, we have converted an aquarium whose elements are natural numbers into an aquarium whose elements are unary functions, but in which the original numbers are just barely hidden under the surface. In fact, it is rather easy to reach under the surface and pull out the number represented by a function, effectively stripping the $[\ ]_+$ annotation.

$$[x]_+(0) = x\ \langle + \rangle\ 0 = x$$

Extracting values from aspects is very useful when assembling aquariums. A piece of plumbing could use this to convert an aspect from one aquarium to another, turning $[x]_+$ into $[x]_\times$, for example. One could also compute an arbitrary expression such as $x^2 + 1$. The point is that while values are annotated, they are aspects which can only be combined using the combination function specified in the index, but once a piece of plumbing extracts a value from an aspect combination, the value can be manipulated arbitrarily.

To extract a value from its unary function form, as we did above, it suffices to apply the function to an identity element, if the combination function has one. Above, the extraction worked because zero is the identity element for $+$.

Another use for identity elements is that once they are annotated, they become identity functions. For example, $[0]_+$, $[1]_\times$ and $[\mathsf{zero}]_{\mathsf{pick\text{-}one}}$ are all identity functions.

Identity aspects are very useful default values when doing aspect composition, since they do not perform any change.

For example, we will see in a moment that bifunctors make it possible to build compound aquariums whose elements change two values at the same time. To build an element that only changes one component, it suffices to use an identity aspect for the component that we do not wish to change.

6.4 Functors

A functor is a type transformer $M : \mathsf{Set} \to \mathsf{Set}$ equipped with a higher-order function $\mathsf{fmap_M}$ which can lift a function f of type $A \to B$ to a function of type $M(A) \to M(B)$. When the underlying functor is obvious from the context, we write $\hat{f}$ for $\mathsf{fmap_M}(f)$. The lifting function must preserve identity and composition, as expressed by the following equations.

$$\widehat{\mathsf{id}}\ x = x \qquad\qquad \widehat{fg} = \hat{f}\hat{g}$$

The fact that $\mathsf{fmap_M}$ must preserve compositions is very handy. If we already have an aquarium whose elements are functions of type $A \circlearrowleft$, we can apply $\mathsf{fmap_M}$ to each element of the aquarium in order to transform it into a new aquarium, containing functions of type $M(A) \circlearrowleft$. Since any pair of functions f and g in the original set are known to commute, we immediately get that their lifted versions $\hat{f}$ and $\hat{g}$ in the transformed set also commute.

$$\hat{f}\hat{g} = \widehat{fg} = \widehat{gf} = \hat{g}\hat{f}$$

There are many, many useful type transformers which happen to be functors: List, Maybe, Tagged, Single, and containers [AAG05] in general. This means that if, for example, the programmer has spent some time constructing an aquarium and later on realizes that she needs some of her aspects to have priority over the others, she can easily upgrade to tagged aspects without losing any of her work proving that her aspects pairwise commute.

6.4.1 Bifunctors

Beyond this large supply of unary functors is an even larger supply of bifunctors $M : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}$, equipped with a higher-order function $\mathsf{ffmap_M}$ having the ability to combine functions f and f' of type $A \to B$ and $A' \to B'$ into a function of type $M(A,\ A') \to M(B,\ B')$. When the underlying bifunctor is obvious from the context, we write $\widehat{f|f'}$ for $\mathsf{ffmap_M}(f, f')$. Again, $\mathsf{ffmap_M}$ must preserve identity and composition.

$$\widehat{\mathsf{id}|\mathsf{id}}\ x = x \qquad\qquad \widehat{fg|f'g'} = \widehat{f|f'}\,\widehat{g|g'}$$

And once again, the fact that $\mathsf{ffmap_M}$ preserves compositions is very handy. This time, we do more than transforming an aquarium: we combine two aquariums into one, allowing aspects to have more than one composable part. For example, as suggested in the previous chapter, one part could be used to describe a procedure, while the other part could set a flag characterizing a property of the procedure, such as whether or not the procedure would benefit from pre-caching.

$$\widehat{f|f'}\,\widehat{g|g'} = \widehat{fg|f'g'} = \widehat{gf|g'f'} = \widehat{g|g'}\,\widehat{f|f'}$$

6.4.2 Multifunctors

Bifunctors generalize to multifunctors of arbitrary arity, whose $\overbrace{\mathsf{f}\cdots}^{n}\mathsf{fmap}$ higher-order functions preserve composition and can thus be used to transform aquariums. This large family of transformations includes tuples — an n-ary version of conjunction — and records, a version of tuples where element positions are given field names.

We use the notation $\{\mathsf{a} : A,\ ...\}$ to denote a record type containing a field named a whose type is A, and the notation $\{\mathsf{a} \mapsto x,\ ...\}$ to denote a record value whose field a has value x. The record type $\{\mathsf{a} : A,\ ...\}$, by itself, does not qualify as a functor because it is only a type, not a type constructor. When we say that the family of multifunctors includes records, we mean that type constructors like the following qualify as functors.

$$M(A,\ ...) = \{\mathsf{a} : A,\ ...\}$$

The $\overbrace{\mathsf{f}\cdots}^{n}\mathsf{fmap}$ function associated with this M takes a number of functions as arguments and applies each one to a field of the input record. To make it more explicit which function is intended to be applied to which field, we write the application of fmap_M to a bunch of functions as $\{\mathsf{a} \mapsto f,\ ...\}_{\mathsf{fmap}}$ instead of $\mathsf{f}\cdots\mathsf{fmap}_M(f,\ ...)$.

$$\{\mathsf{a} \mapsto f,\ ...\}_{\mathsf{fmap}}\Big(\{\mathsf{a} \mapsto x,\ ...\}\Big) = \{\mathsf{a} \mapsto f(x),\ ...\}$$

Records can be used to make aquariums much more intelligible. For example, let's define an aquarium solving a typical scenario involving mutually recursive modules. Our goal is to define two modules, Even and Odd, each containing a function from Nat to Bool calling the other module's function as a helper.

Clearly, this is something we can already accomplish using non aspect-oriented modules, so this does not serve to demonstrate that aquariums solve a new problem. Rather, we solve an old problem in a new way, to familiarize ourselves with our new tools.

If we were trying to implement both even and odd in the same module, we could simply use a mutually recursive primitive such as Scheme's **letrec** or Agda's **mutual**, and define each function as an expression of type Nat → Bool. Since we want to define them in separate modules, however, we must use additional parameters giving each function an access to the other.

```
Even-Odd = {even : (odd  : Nat → Bool) → Nat → Bool,
            odd  : (even : Nat → Bool) → Nat → Bool}
```

The idea is that once we combine our modules, we obtain a record of the form $\{\mathsf{even} \mapsto x,\ \mathsf{odd} \mapsto y\}$ containing an implementation for each function. A piece of plumbing can then use mutual recursion to link the implementations into single-parameter versions of even and odd.

```
link : Even-Odd → (Nat → Bool) ∧ (Nat → Bool)
link even-odd = (rec-even, rec-odd) where
  open Even-Odd even-odd
  mutual
    rec-even = even rec-odd
    rec-odd  = odd rec-even
```

In the link implementation above, even-odd is a record value, and **open** makes its fields even and odd available within the **where** block. **mutual** is necessary in

order for the definition of rec-even to refer to rec-odd. This link function, however, is only useful once the record is fully constructed. To do this, let's define an aquarium allowing each aspect to provide one of the two required implementations.

First, notice that we are not able to link if we miss either implementation, and that the linking is ambiguous if we have more than one aspect providing an implementation for one of the two functions. We require exactly one of each, and thus, we use the functor Single, instantiating X with $(\mathsf{Nat} \to \mathsf{Bool}) \to \mathsf{Nat} \to \mathsf{Bool}$.

More precisely, we instantiate $\lambda(A, B).\ \{\mathsf{even} : \mathsf{A},\ \mathsf{odd} : \mathsf{B}\}$ with the arguments $A = B = \mathsf{Single}$. Since the record type constructor is a multifunctor, and we already know that $[\![\mathsf{Single}]\!]_{\mathsf{pick\text{-}one}}$ is an aquarium, we can conclude that the functions $\{\mathsf{even} \mapsto [\mathsf{one}\ x]_{\mathsf{pick\text{-}one}},\ \mathsf{odd} \mapsto [\mathsf{zero}]_{\mathsf{pick\text{-}one}}\}_{\mathsf{fmap}}$ and $\{\mathsf{even} \mapsto [\mathsf{zero}]_{\mathsf{pick\text{-}one}},\ \mathsf{odd} \mapsto [\mathsf{one}\ y]_{\mathsf{pick\text{-}one}}\}_{\mathsf{fmap}}$ commute with each other, and with many other lifted functions like them.

Those two example functions contain one implementation each (the arguments to the one constructors), and can be placed in separate modules if desired. They are also intelligible, since the field names clearly mark the purpose of each component. The purpose of these aspects could be made even more clear by using the synonyms import and export instead of zero and one.

Once the aspects are composed, the result should be $\{\mathsf{even} \mapsto [\mathsf{one}\ x]_{\mathsf{pick\text{-}one}},\ \mathsf{odd} \mapsto [\mathsf{one}\ y]_{\mathsf{pick\text{-}one}}\}_{\mathsf{fmap}}$, which is a record-manipulating function, not a record. Applying it to $\{\mathsf{even} \mapsto \mathsf{zero},\ \mathsf{odd} \mapsto \mathsf{zero}\}$, the identity element of the type $\{\mathsf{even} : \mathsf{Single},\ \mathsf{odd} : \mathsf{Single}\}$, is the usual technique to collapse the function into a value, which should be equal to $\{\mathsf{even} \mapsto \mathsf{one}\ x,\ \mathsf{odd} \mapsto \mathsf{one}\ y\}$.

This value would contain zero or many values as error markers if the number of contributions were incorrect, and the linking function needs to be updated slightly to account for this fact.

This example of mutually recursive modules demonstrates that even though aquariums are designed to separate commutative implementation fragments into different modules, the programmer can also use them to separate traditional, non commutative implementation fragments such as even and odd, using the Single type constructor and a bit of plumbing.

The advantage of doing it this way, instead of using a more traditional linking process, is that once the code is structured to use aquariums, it is easy to adapt it to use other aquariums. Thus, the original program can start with non commutative parts, and its aquariums can be gradually refined as the need for new aspects arises.

This refinement can be achieved by replacing Single with one of the many aquariums presented in the previous chapter, or, if none of them are sufficient, a new one can be created using the theorems of this chapter.

CHAPTER 7
Conclusion

This thesis offers numerous ideas, proofs and implementation strategies to help the programmer avoid conflicts in her aspect-oriented programs. In terms of avoiding conflicts, programming with aquariums is an improvement over existing aspect-oriented languages, because they free programmers from the need to constantly check their work with that of their colleagues to see if a conflict has arisen.

7.1 Future Work

Despite the improvements, there is still more work to be done before the ideas of this thesis can be used concretely to build real-world applications using an aspect-oriented language approaching the current state of the art.

This section suggests many ways in which this work could be extended in order to make it more useful.

7.1.1 Example Programs

Commutativity imposes hard limits on which functions can be used together and which cannot. This naturally leads to a very important question: is the space of commutative functions expressive enough to describe the kind of aquariums programmers want to write, or will most of the variability be confined to the plumbing?

The next step, to figure out the answer to this question, is to implement larger examples using our Agda prototype. This would also allow us to discover the rough edges in our design, and to test its suitability for practical programming.

There is one program, in particular, which will reveal a lot of information about our technique when we will implement it. This program, perhaps surprisingly, is GNU sort.

The GNU sort program has been reimplemented in a few different aspect-oriented languages [Car02], with the goal of comparing the suitability of the languages for this task. Since our goal has always been to solve conflicts, not to improve on the expressivity and modularity of existing languages, it would be interesting to see how the designs that follow from using aquariums compare, in those respects, to the existing implementations.

7.1.2 Less Rewriting

A major obstacle preventing many stakeholders from jumping on moving technological bandwagons is their existing assets, which they would need to adapt to the new technology. In this respect, AspectJ is quite impressive since it managed to shoehorn a major new paradigm, aspect-oriented programming, on top of a popular real-world language, Java. Given the unwarranted resistance that AspectJ is already facing, it is clearly not realistic to expect businesses to switch to Agda in order to benefit from our technology.

Fortunately, aquariums are not a completely new approach, but a generalization of existing composition mechanisms. In section 6.4.2, we have shown how the Single

combinator can be used to forbid the implementation of a single function, such as even or odd, to be distributed among many aspects. An aquarium using this strategy can be considered degenerate, since it doesn't allow multiple implementations to be composed. The upside is that since there is only one implementation per function, there is no need to check that this implementation commutes with anything else. Therefore, these degenerate aquariums correspond to traditional module-based composition, where having multiple implementations for a single function is an error and where commutativity proofs are not required.

Therefore, it would theoretically be possible to follow AspectJ's lead and shoehorn our ideas to an existing language. This would be a lot of work, but it would be very beneficial, since it would make it possible for actual programmers working on real-world projects to benefit from the technology.

7.1.3 Less Proofs

Working in a functional language which doubles as a proof assistant was very convenient for us, but we can imagine how the casual programmer might be frightened at the prospect.

For this reason, it would be highly beneficial to select those of our proofs that can be applied in a type-directed way, and to write a language in which the type checker took care of ensuring commutativity. The technique shown in chapter 5, for example, is mostly automated, relieving the programmer from the task of writing commutativity proofs. Since the automation is provided as a library, however, the programmer still needs to write some boilerplate code in order to benefit from the

automation. A language with a built-in concept of aquariums could do away with such boilerplate.

7.1.4 Associativity

Some proofs in this thesis require both commutativity and associativity, but only commutativity was examined in details. An obvious next step is to find rules for writing intrinsically associative functions, if such rules exist.

Here are some of our current insights regarding associativity.

Conjunction

If a datatype has only one constructor, then fmap preserves its associativity. Here is the proof for pairs, for example, using the syntax for bifunctors presented in section 6.4.1.

The proof shows that two associative functions $(\circ)$ and $(\bullet)$, when combined using fmap into a function $(\widehat{\circ|\bullet})$ that applies $(\circ)$ to its left parameters and $(\bullet)$ to its right parameters, continue to be associative. All three functions are written infix.

$$\Big((x,u)\ \widehat{\circ|\bullet}\ (y,v)\Big)\ \widehat{\circ|\bullet}\ (z,w) = (x \circ y \circ z, u \bullet v \bullet w) = (x,u)\ \widehat{\circ|\bullet}\ \Big((y,v)\ \widehat{\circ|\bullet}\ (z,w)\Big)$$

Notation

The prefix notation for associativity, $f\ (f\ x\ y)\ z = f\ x\ (f\ y\ z)$, looks notoriously different from the infix version usually given. If we use section 6.3's notation for

partial application, however, we obtain a version which doesn't look as awkward.

$$\begin{aligned}
\forall x\ y\ z.\quad (x+y)+z &= x+(y+z)\\
\forall x\ y\ z.\quad [x+y]_+(z) &= ([x]_+[y]_+)(z)\\
\forall x\ y.\qquad\quad [x+y]_+ &= [x]_+[y]_+
\end{aligned}$$

Associative functions, it would seem, are those functions for which applying the result of $x+y$ is the same as applying y, then x. Hopefully, just as the infix notation is much more inspiring than the prefix notation, this new way of looking at associativity might offer insights which the original notation did not.

Subsets Closed Under Function Composition

One new insight offered by this new notation for associativity is the link between associative functions and families of functions closed under function composition.

If two types A and B are isomorphic to each other and an associative function is defined for A, it is straightforward to convert this function into an associative function on B. For example, if A is actually the function space $C\ \circlearrowleft$, function composition can be converted into an associative function on B.

A slightly more interesting case occurs when A is a subset of $C\ \circlearrowleft$. In this case, the subset may or may not be closed under function composition, because it is possible for a composed function to lie outside of the subset. In the case that the subset is closed, function composition restricted to this subset is well-defined, and thus corresponds to an associative function on B. Furthermore, any associative function corresponds to such a closed subset.

To see this, consider an arbitrary binary function on type B, denoted $+$, known to be associative. Next, construct a subset of the unary functions of type $B \circlearrowleft$ consisting of those functions for which there exists a value b of type B such that the function can be written as $[b]_+$. With our new notation for associativity, it is straightforward to see that this subset must be closed under function composition: given any two functions $[x]_+$ and $[y]_+$ drawn from the subset, there exists a value b such that the composition $[x]_+[y]_+$ can be written as $[b]_+$, namely, $b = x + y$.

7.1.5 Category Theory

Requiring function f to commute with function g is an unusual requirement in computer science, but it is routine for mathematicians studying category theory [ML98]. For example, category theory defines a natural transformation from F to F to be a function η mapping every object A to a morphism $\eta_A : F\ A \to F\ A$ such that, among other things, any lifted morphism $\hat{f} : A \to A$ commutes with η_A.

Our guess is that reinterpreting the ideas of this thesis in the language of category theory could yield many useful insights, and could allow us to reuse many existing proofs from category theory.

7.1.6 Commutative Functors

In this thesis, we have examined how to create aquariums, how to extend aquariums, how to combine aquariums, but we have never considered the possibility of composing aquariums. Here is a situation where this might be handy.

The record example in section 6.4.2 highlights a very common situation, where each module implements only one or a few fields, assigning id (in the form of

$[\mathsf{zero}]_{\mathsf{pick\text{-}one}}$) to all others. It would be nice if those fields could be added automatically. Doing so would involve examining the fields of the aspects which need to be composed, combining them all into a single record type, and extending all aspects to match that record type.

In other words, doing this would involve combining aquariums, and converting each aspect from its source aquarium to the combined aquarium.

We haven't worked out all of the details yet, but commutative functors point to a convenient way to do this.

Suppose we have two functors, F and G, both of type $\mathsf{Set}\circlearrowleft$. Because of their types, those two functors can be composed in either order. If we assume that the compositions FG and GF both result in the functor H — that is, if we assume that the functors commute — then $H(A)$ is a sane choice for the combination of the two aquariums $F(A)$ and $G(A)$.

Moreover, the fmap functions associated with F and G can be used to lift functions of type $F(A)\circlearrowleft$ and $G(A)\circlearrowleft$ to functions of type $H(A)\circlearrowleft$. This is useful because H's own fmap function can only lift functions of type $X\circlearrowleft$ to functions of type $H(X)\circlearrowleft$, which would yield functions of type $HF(A)\circlearrowleft$ and $HG(A)\circlearrowleft$ instead of $H(A)\circlearrowleft$. These various fmap functions are the primitives which should be used to convert each aspect from its source aquarium to the combined aquarium.

$$\begin{aligned}
\mathsf{fmap}_H &: (\ \mathsf{A}\ \to\ \mathsf{B}\ ) \to\ \mathsf{H\,A}\ \to\ \mathsf{H\,(B)} \\
\mathsf{fmap}_F &: (\mathsf{G\,(A)} \to \mathsf{G\,(B)}) \to \mathsf{F\,(G\,(A))} \to \mathsf{F\,(G\,(B))} \\
\mathsf{fmap}_G &: (\mathsf{F\,(A)} \to \mathsf{F\,(B)}) \to \mathsf{G\,(F\,(A))} \to \mathsf{G\,(F\,(B))}
\end{aligned}$$

7.1.7 Joinpoints

From the start, we have decided to ignore joinpoints and to focus only on composition. This has led us to a design involving aquariums, where each aquarium has only one joinpoint on which all aspects are applied. Can we adapt this design to a language that uses joinpoints and pointcuts, and in which advice are woven at various joinpoints in the program's execution?

As a preliminary result, we can already establish that aspects whose pointcuts do not intersect cannot conflict. Indeed, when calls are weaved at distinct points of the program, the resulting code must be the same regardless of the weaving order. More importantly, the behaviour of the resulting program must be the same regardless of the weaving order, so two aspects whose joinpoints do not intersect must commute.

Our usual argument is easily applied to this new context. We assume that each aspect works, that is, that weaving this single aspect into any program always results in a program satisfying the aspect's target property. Next, we observe that if we weave aspect A before aspect B, it must be the case that the resulting program satisfies B's target property. Next, we observe that weaving the aspects in the opposite order results in a program satisfying A's target property. Finally, we use commutativity to point out that the two weaving orders result in identical programs, and this resulting program thus satisfies both target properties. Therefore, the aspects do not conflict. □

When pointcuts are shared between more than one aspect, the problem becomes harder. It seems like we should validate that the actions weaved at the common

joinpoints commute with one another. But how to ensure this? How would we even specify the aquarium to be used at each joinpoint? We could use a pointcut to associate an aquarium with a set of joinpoints, but in this case, what happens if the programmer attempts to associate more than one aquarium with the same joinpoint?

This line of research is rich with open questions.

7.1.8 Imperative Code

There is another ubiquitous feature of existing aspect-oriented languages which has been conspicuously missing from our presentation: the possibility of side effects.

One obvious solution to handle side effects is to include the world state in the definition of commutativity. Two effectful functions f and g could be said to commute if, for any input x and any starting world state w, the applications $fg(x)$ and $gf(x)$ return the same result and move the world to the same world state.

Another approach would be to use Haskell [Jon03]'s IO monad to represent effectful functions as pure functions of type $\mathsf{A} \to \mathsf{IO}\ \mathsf{A}$. Note that function of this type are not endofunctions, and thus cannot be composed commutatively using function composition. We can, however, combine two such functions f and g using the Kleisli composition operator, $(>=>)$, of type $(a \to M\ b) \to (b \to M\ c) \to (a \to M\ c)$. With this modification, f and g could be said to commute if, for any input a, the applications $(f >=> g)(a)$ and $(g >=> f)(a)$ return equal results.

A problem with both of these approaches is that in practice, effectful functions are often blind calls to C procedures, whose effects on the outside world are not modelled precisely. Functional languages often avoid the issue by simply asserting

that effectful functions always create fresh, never seen before world states. The fact that these states are fresh, however, prevents two such states obtained through different means from being equal. This approximation is very useful in practice, but it defeats our purpose since it ensures that no two effectful functions can ever be considered commutative.

7.1.9 Object-Orientation

Tutorials introducing aspect-orientation are usually very clear on the subject: aspects are not meant to replace, but to complement objects. In our case, objects complement our approach very well, since they provide natural boundaries for our aquariums. Here are two sketches describing how this could be done.

Object-Level

At the level of a single object, we could view the different methods of that object as the different actions allowed by the aquarium. Some specially marked objects would ensure that the result of calling several of their methods would always result in the same internal state, regardless of the order in which their methods were called.

Then, within advices, the compiler would make sure that only methods from commutative objects were called. Thanks to the functor transformation rule, the conjunction of all the states of the commutative objects is also an aquarium, so an advice could be seen as an aspect in a bigger aquarium.

Class-Level

Alternatively, we could view the object behaviour as the end result, to be constructed by the commutative collaboration of several aspects. Each class could decide to define a new function, or to refine an existing one defined in a superclass. Each refinement would be performed according to the aquarium which was chosen when the function was first defined.

Following CaesarJ [MO03], the same could be performed at a bigger level, to construct and compose classes and "cclasses" (classes whose members are other classes, not methods) using higher-order aquariums. The resulting nesting of aquariums would be very interesting to study.

7.2 Summary

The goal of this thesis was to free the programmer from the need to reason about the possibility of conflicts, by delegating the task of detecting conflicts from the programmer to the compiler.

We have accomplished this goal by reducing the need to reason about aspect interactions to the simpler need to reason about commutativity. Reasoning about commutativity is simpler than reasoning about conflicts, because commutativity is a well-defined concept which stands on its own, whereas the notion of conflict requires the target property of each aspect involved to be found, written down, and checked. We have shown that commutativity is sufficient to avoid conflicts by proving that regardless of the target properties which would have been written down, commutative aspects which work well in isolation continue to satisfy those properties after they are composed together.

The compiler's job, in our solution, is not to check commutativity but to check the validity of commutativity proofs provided by the programmer. To assist the programmer in this task, we have proved a number of short but useful theorems which can be used by the programmer as lemmas. In addition, to reduce the need for proofs, we have described a technique for writing intrinsically commutative functions.

We have divided our work into two kinds of commutativity guarantees, unary and binary commutativity, as each of them is useful in a different situation. Binary commutativity is useful for custom aspect representations, when the number of allowed aspect actions needs to be controlled, while unary commutativity is useful in the common case, when the aspects are represented as functions.

In both situations, the aspects are clustered into subsets of compatible aspects, called aquariums. This organization allows the decisions leading to a conflict-free environment to be taken independently of the choice of which aspects to include, a division which could be an advantage if some programmers in a team are more knowledgeable than others in the domain of aspect interactions. Separating aspect design from aquarium design could be a very useful technique to manage the complexity of aspect interference, regardless of whether commutativity is also used.

It is our hope that both ideas, commutativity and aquariums, have convinced the reader that avoiding conflicts is possible. If our ideas end up being used to solve concrete conflicts between aspects in a real-world project, we would be delighted to hear about it. Who knows? Maybe someone will be sufficiently inspired by the fact

that conflicts, in the setting of aspect-orientation, can be solved, to attempt to solve other kinds of conflicts, in a different domain.

Thank you for reading.

References

[AAG05] Michael Abott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science*, 342:3–27, September 2005.

[All09] All papers in this issue. *Transactions on Aspect-Oriented Software Development: Special Issue on Dependencies and Interactions with Aspects*. Springer, 2009.

[BC90] Gilad Bracha and William Cook. Mixin-Based Inheritance. *ACM SIGPLAN Notices*, 25(10):303–311, 1990.

[Bec03] Kent Beck. *Test-Driven Development: By Example*. Addison-Wesley, 2003.

[Car02] Lee Carver. Composition Behaviors for Application Construction. *Workshop on Aspect Oriented Design, First International Conference on Aspect Oriented Software Development*, 2002.

[Cos03] Pascal Costanza. Dynamically Scoped Functions as the Essence of Aspect-Oriented Programming. *ACM SIGPLAN Notices*, 38(8):29–36, 2003.

[EAK+01] Tzilla Elrad, Mehmet Aksit, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing Aspects of Aspect-Oriented Programming. *Communications of the ACM*, 44(10):33–38, October 2001.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification*. Addison-Wesley Professional, 2005.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[KBdR99] Gregor Kiczales, Daniel G. Bobrow, and Jim des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1999.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–357, 2001.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis Napoli, 1984.

[ML98] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1998.

[MO03] Mira Mezini and Klaus Ostermann. Conquering Aspects with Caesar. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 90–99, 2003.

[Nor09] Ulf Norell. Dependently Typed Programming in Agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, pages 1–2, 2009.

[Par72] David Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1058, 1972.

[RA07] André Restivo and Ademar Aguiar. Towards Detecting and Solving Aspect Conflicts and Interferences using Unit Tests. In *Proceedings of the 5th Workshop on Engineering Properties of Languages and Aspect Technologies*, 2007.

[Rou05] David Roundy. Darcs: Distributed Version Management in Haskell. In *Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 1–4, 2005.

[Ser14] François Servois. Essai sur un Nouveau Mode d'Exposition des Principes du Calcul Differentiel. *Annales de Mathématiques*, 1814.

[Ste90] Guy L. Steele. *Common LISP: The Language*. Digital Press, 1990.

[Sup72] Patrick Suppes. *Axiomatic Set Theory*. Dover Publications, 1972.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings of the 4th International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, 1989.

[WG00] David Wagner and Ian Goldberg. Proofs of security for the Unix password hashing algorithm. *Lecture notes in computer science*, pages 560–572, 2000.

[WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A Theory of Aspects. *ACM SIGPLAN Notices*, 38(9):139, 2003.