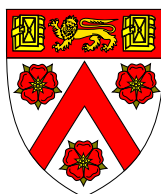




Programming and static analysis with graded monads

Andrej Ivašković



Trinity College
University of Cambridge
9 January 2022

This dissertation is submitted for the degree of Doctor of Philosophy.

Declaration

This dissertation is my own work and includes nothing which is the outcome of work done in collaboration with others except as declared in the preface and specified in the text. It is not substantially the same as any work that has already been submitted before for any degree or other qualification except as declared in the preface and specified in the text. This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.

Abstract

Programming and static analysis with graded monads

Andrej Ivašković

Programmers writing purely functional code have long represented impure, side-effecting computation using monads. Recent research in program semantics has focussed on graded monads, a useful generalisation of monads which allow the programmer to establish useful properties of a computation purely from their type of a computation. They have been used to represent a variety of properties traditionally inferred by frameworks such as type-and-effect systems – for example, success of partial computation and taint checking. However, their use has been limited to defining the formal semantics of type-and-effect systems and to embedding effect systems into pure functional languages (such as Haskell). Graded monads have not previously been seen as an interesting programming construct on their own. This thesis explores the capability of graded monads to represent a variety of program analyses in a unified fashion, as well as the extent to which programmers are able to define their own interesting program analyses.

This thesis first defines the class of analysis-graded monads, a class of graded monads whose intended use is to represent what is commonly thought of as ‘program analysis’ or ‘effect system’. A contribution of this thesis is a zoo of examples of analysis-graded monads, as well as their representation in Haskell. Analysis-graded monads are considered from both the programming side and as a tool for defining a denotational semantics (using category theory). The thesis considers the matters of type inference in language with support for graded monadic types, effectively framing program analysis as a question of type (and grade) inference. A practical implementation comes in the form of a Haskell package (`analysis-monad`).

The remaining contributions of the thesis address the use of analysis-graded monads in more sophisticated settings and large programs. Firstly, I consider computations and associated analyses that can be seen as being parameterised by a set of distinct components (such as individual exception names forming an analysis similar to Java checked exceptions), and propose the concept of *graded monad builders* as a way of generating such analysis given a set of such component names. Secondly, I look at the question of combining more than one analysis, which effectively means combining the role of several analysis-graded monads.

This research on programming with graded monads and their use in program analysis informs conclusions about the way program analysis for effectful computation can be seen as an instance of typing.

Acknowledgements

Writing this thesis and completing my PhD was enjoyable: the topic was intellectually stimulating and I had the opportunity to collaborate with amazing people. The journey was not without its bumps – the most notable of which was the rude interruption by the COVID-19 pandemic halfway through my PhD. Thankfully, there were many people that helped me get this thesis done on time to whom I owe a lot of gratitude.

First, I must acknowledge and commend Alan Mycroft, my supervisor. Alan was a great guide who helped me find a story to tell in this thesis and complete this PhD. Other than being knowledgeable about everything, Alan had a very positive and encouraging attitude, as well as the will to read my barely comprehensible writeups of various ideas and dilligently comment on them – even when I sent them way past when I initially said I would. This thesis, as well as all of the papers I co-authored with him, would not be readable without his input. He gave me valuable advice regarding getting work done that I will hold long after this PhD. I will miss our long discussions about research, which exposed me to ideas that went beyond the narrow focus of whatever was the original topic. I feel I have a much better ‘big picture’ view of language design and type systems thanks to these discussions.

I also received a lot of help and guidance from Neel Krishnaswami, who acted as a second supervisor. He was happy to arrange weekly meetings with me to discuss my progress and research, as well as comment on writeups. I found it especially useful to get his feedback on the parts of my research that delved into defining formal semantics. I also attended weekly meetings with Neel’s group of students in which we discussed our progress and had the opportunity to ask each other clarifying questions and recommend additional references. Among the many group members, I will mention Michael Arntzenius, Vikraman Choudhury, Faustyna Krawiec and Dhruv Makwana as particularly friendly and curious colleagues.

I have all the best words and thoughts about the Programming, Logic and Semantics group at the Department of Computer Science and Technology, especially the PhD students in there. The PhD student community in the group was tightly-knit and willing to organise various activities, such as helping each other with preparing for undergraduate supervisions or discussing research topics. I found the Logic and Semantics for Dummies seminars very valuable because they both provided me with knowledge I needed in my PhD and experience presenting my work in a seminar environment. The (pre-pandemic) daily lunches and Theory Tea were a great opportunity to make friends and get support with work. I am thankful for the support and friendship with Nathanael Arkor, Shaun Steenkamp, Ian Orton, Matthew Dagitt, Hugo Paquet, Chelsea Edmonds, Adam O’Conghaile, Dmitrij Szamozvancev, Dylan McDermott and others.

While the pandemic made going to the office more difficult, I have all the best words for my two office-mates: Christopher Elsby and Indigo Orton. I only wish I had seen more of them in the second half of my PhD.

I am thankful to Dominic Orchard, my very dilligent and gracious co-author.

I am also grateful to Trinity College, who provided funding for my PhD via the Internal Graduate Scholarship.

I am thankful to the anonymous referees for providing feedback to my papers submitted to various conferences.

I am grateful to my examiners, Jeremy Yallop and Sam Staton, who provided valuable feedback that made this thesis better. They made the viva a very pleasant experience and an opportunity to present the three years' worth of PhD research.

I appreciate the continuing support and encouragement given by my family: my mum Milka, my dad Boban, and my brother Stefan. At every point in my PhD they were curious about how I was doing and they paid close attention to conference deadlines and all feedback I received. I am thankful for their words of encouragement and for believing in me. Our Samoyed Nanuk deserves a mention as well for being a joy to work next to.

Finally, I am thankful to Aga, my love and soul, my life partner. She has been with me through it all and encouraged me to not give up. I am glad that I spent most of the pandemic and lockdown close to her, including holidays with her very helpful and supporting mum Iza and brother Andrzej. I am grateful for everything that makes Aga the wonderful person she is.

Contents

I Preliminaries	1
1 Introduction	3
1.1 Programming with effects	3
1.2 Program properties	6
1.3 Contribution	7
1.4 Thesis structure	8
2 Background	11
2.1 Analysis frameworks	11
2.1.1 Lattice theory	11
2.1.2 Type-and-effect systems	14
2.1.3 Abstract interpretation	17
2.2 Programming with graded monads	20
2.2.1 Monads	20
2.2.2 Graded monads	24
2.3 Category-theoretic view	27
2.3.1 Monads and graded monads	27
2.3.2 Categories associated with monads	31
2.3.3 Monad transformers as monoid transformers	32
2.4 Related work	33
2.4.1 Effects	33
2.4.2 Comonads and coeffects	36
2.4.3 Related monad generalisations	37
II Analysis-graded monads	39
3 Program analysis through grades	41
3.1 An analysis-graded monadic framework	42
3.1.1 Defining Target	42
3.1.2 Programming with analysis-graded monads	44
3.1.3 Haskell embedding	52
3.2 Computing with failure	54
3.2.1 Deciding safety and failure	54
3.2.2 Checked exceptions	58
3.3 Stateful computation	61
3.3.1 Permissions	61

3.3.2	Live variable analysis	64
3.3.3	Logging updates	70
3.4	Safe concurrency	73
3.4.1	Avoiding deadlock with structured locking	74
3.4.2	Graph-based analysis	77
3.4.3	Related work	80
3.5	Discussion and limitations	81
3.6	Summary	82
4	Grade inference	83
4.1	Grade holes	83
4.1.1	A gradeless Source variant	84
4.1.2	Analysis-graded monads with $\perp$	86
4.1.3	The grade inference framework	87
4.1.4	Solving constraints	91
4.1.5	Adding recursion	93
4.2	Polymorphic grades	94
4.2.1	Grade-polymorphic programming	94
4.2.2	Grade inference	97
4.2.3	Discussion	99
4.3	Summary	99
5	Semantics of analysis-graded monads	101
5.1	Categorical view	101
5.1.1	Monad at the top	102
5.1.2	Category of analysis-graded monads	103
5.1.3	Abstractions	105
5.2	Categorical semantics	106
5.2.1	Semantics of Target	106
5.2.2	Semantics of Source	109
5.3	Summary	112
III	Adding structure	113
6	Graded monad builders	115
6.1	Towards builders and symbols	116
6.1.1	Examples and syntax	116
6.1.2	Definitions	118
6.2	System G: polymorphic symbols and grades	118
6.2.1	Syntax and types	118
6.2.2	Denotational semantics	122
6.2.3	Equational theory	125
6.3	ML_G : inferring polymorphic grades	125
6.3.1	Syntax and types	126
6.3.2	Type inference	129
6.3.3	Relationship with System G	131
6.4	Summary	134

7	Combining analyses	135
7.1	Graded monad transformers	136
7.1.1	Haskell view	137
7.1.2	Categorical definition	139
7.1.3	Properties	143
7.1.4	Semantics of lifting	148
7.2	Grade families	149
7.2.1	Haskell embedding	150
7.2.2	Formal definition	151
7.3	Discussion and further work	152
7.4	Summary	153
IV	Concluding remarks	155
8	Language design	157
8.1	Present limitations and difficulties	157
8.2	The Granalyse language	158
8.2.1	A two-level language	159
8.2.2	Recursion	160
8.2.3	Symbols and builders	161
8.3	Summary	161
9	Conclusion	163
9.1	Summary of contributions	163
9.2	Further work	163
9.2.1	Other graded structures	164
9.2.2	Further theoretical results	164
9.2.3	An implementation of Granalyse	165
V	Appendices	175
A	Index of notation	177
A.1	Mathematics	177
A.1.1	General	177
A.1.2	Category theory	177
A.2	Language syntax	179
B	Category-theoretic background	181
B.1	Categories	181
B.1.1	Definition of a category	181
B.1.2	Functors and natural transformations	182
B.1.3	Bicategories	183
B.2	Universal constructions	184
B.2.1	Initial and terminal object	184
B.2.2	Products and coproducts	184
B.2.3	Exponentials	185
B.2.4	Limits and colimits	186
B.3	Monoidal categories	186

B.3.1	Laws and classification of monoidal categories	186
B.3.2	Monoids	187
B.3.3	Monoidal functors and natural transformations	188
C	Haskell code listings	191
C.1	Analysis-graded monads	191
C.1.1	Usage example	191
C.1.2	Components of analysis-graded monads	191
C.1.3	Stateful computation	193
C.1.4	Other analysis-graded monads	200
C.2	Graded monad transformers	203
C.2.1	Definition	203
C.2.2	Instances of transformers	203
C.3	Graded concurrency	211
C.3.1	Graded monad definition	211
C.3.2	Usage examples	215
D	Denotational semantics of System G	219
D.1	Overview of denotational semantics	219
D.2	System G: full denotational semantics	221
E	Extended examples	227
E.1	Live variables	227
E.2	History-graded state monad transformer	227
F	Additional proofs	233
F.1	Proof of soundness of the System G equational theory	233
F.2	Proof of correctness of type inference for ML_G	234

Part I

Preliminaries

This part of the thesis sets the stage for the main contribution of the thesis: representing program analysis through the type system of a language using *graded monads*. The aim is to acquaint the reader both with the idea of program analysis and programming with monads and their graded generalisation.

The introductory Chapter 1 provides the context of the thesis contribution. In Chapter 2 I give the relevant mathematical and functional programming background for discussing graded monads and programming with them.

Introduction

My thesis looks at the ways in which program analyses concerned with side-effect analysis in programming languages can be represented through the type system in pure functional languages using graded monads, as well as the language features that can help in this endeavour.

The notion of data type is central to modern programming languages. At the very least, the type of a term carries information about what kind or shape of data it is – integer, string, boolean, binary tree. A static type system can be seen as placing additional constraints on all valid programs, catching errors beyond the trivial syntactic ones and rejecting some incorrect programs. For example, a Standard ML programmer might write the following definition:

```
fun wrong x = "foo" ^ (x + 1)
```

and get a type error, since $x + 1$ is necessarily an integer, and the string concatenation operator `^^` cannot have a non-string operand.

Research in type systems has long looked at the kinds of properties that can be expressed with a suitable type system, going beyond shape and also looking at behaviour of computations. In this thesis, I consider type systems that represent properties associated with side-effecting computations. Such type systems are commonly referred to as *type-and-effect systems*. More precisely, the focus is on pure functional languages (such as Haskell) in which side-effects are represented using monadic types. The goal of the thesis is to explore linguistic features and type systems that make reasoning about computational effects convenient for the programmer, with the aim of representing the task of program analysis as typing.

1.1 Programming with effects

A computational effect refers to any way a program affects its environment. I/O is clearly one such example, as it makes the program interact with the operating system or a device. Raising an exception is also an example, as upon the raise the call stack gets unwound until a handler for the exception is reached. Reading from and writing to a global mutable variable is also a computational effect, as their values form the environment and affect the semantics of other parts of the program. Computational effects in an imperative language break purity.

A persistent canard against functional programming languages is that they are ill-suited for writing any ‘serious’ code that must use side-effects: working with mutable variables with explicit assignments is discouraged as they may prevent useful compiler optimisations; side-effects break the idea that a function in a programming language corresponds to a mathematical partial function. There are two traditional approaches functional languages use to address computational effects:

adding explicit linguistic constructs for these effects (such as reference types and exceptions) and by using *monadic types* with *bind* and *return* primitives. The former approach is taken by Standard ML and OCaml, whereas the latter is the idiomatic way of representing effects in Haskell. The former makes the languages impure and breaks *referential transparency*: the body of `let x = e in e'` cannot use `x` and `e` interchangeably and maintain semantic equivalence. Adding such explicit effects into the language requires adding restrictions to parametric polymorphism in the language in order to preserve soundness – this is usually called the value restriction in let-expressions. By contrast, programming with monads does not require the programmer to sacrifice these language features, but it does lead to a difference in how they are treated at the type level. While an OCaml expression of type `int` might have a side-effect that is not exposed in its type, a Haskell expression of type `Int` cannot have any side-effects and an I/O side-effecting expression that eventually returns a number instead has type `IO Int`.

Type-and-effect systems

In ML-like languages where effects have explicit linguistic constructs (such as assignment), reasoning about computational effects is traditionally done using *type-and-effect* systems. Type-and-effect (sometimes referred to as just *effect systems*) systems were first introduced by Lucassen and Gifford [LG88] in the setting of reasoning about access to mutable state. A formal treatment of a simple language's type system is built on the notion of a typing judgement of the form $\Gamma \vdash M : A$, where Γ is a typing context, M is a term and A is a type, stating that an expression M has type A given a context Γ of variable typing. A type-and-effect system takes this idea a step further to ML-like languages and considers judgements of the form $\Gamma \vdash M : A \ \& \ F$, where F is a representation of the computational effects of M . We say that effect systems *delimit* the scope of computational effects. Type-and-effect systems have been used to reason about a wide variety of computational effects, including region-based memory accesses [TJ94] and concurrency by message passing [NN94].

Graded monads

Both monads and type-and-effect systems delimit computational effects. It is therefore unsurprising that there is a link between monads and effects, first identified by Wadler and Thiemann in their 'Marriage of monads and effects' paper [WT03]. They showed how type-and-effect judgements in an ML-like language with support for mutable stores can be translated into typing judgements in a pure call-by-name language with a monad representing stateful computation.

The work of Wadler and Thiemann exposed that the marriage of effects and monads is an uncomfortable one. A monad does not contain any information other than what is the potential kind of side-effect – such as which variables might be read from or written to in a stateful computation, just that it is stateful. They assumed a family of type constructors `Statee`, with e ranging over all possible effects, as opposed to a single `State` monad. Furthermore, the typing rules in their CBN language do not match the ones expected of monadic operations (*bind* and *return*). Since then, the structure they described has been recognised as a *graded monad*, also sometimes referred to as a *parametric effect monad*.

Informally, a graded monad T is defined over a grading algebra $(E, \odot, 1)$, which is at least a monoid with binary operator $\odot$ and identity 1 . Then a graded monad

T is a family of type constructors T^e in which the grade $e \in E$ such that an expression of type $T^e A$ represents a computation with side-effects represented by T that eventually returns a value of type A (if it halts), where F corresponds to the effect of the expression. That way a program property is represented as *part of the type of an expression*.

The two fundamental operations of a graded monad are *bind* (or *sequencing*) and *return*, just like with monads. A bind of two terms M and N of types $T^e A$ and $A \rightarrow T^f B$ (respectively), for $e, f \in E$, is an expression $M \gg= N$ whose type is that is also graded monadic, but the grade of the resulting type uses the underlying multiplication operator $\odot$ of A – hence the resulting type is $T^{e \odot f} B$ and the grade represents the outcome of sequencing two effectful expressions. The grading algebra E corresponds to the space of effects in the type-and-effect system, which is why it is sometimes referred to as an effect algebra.

Programming with graded monads

In practice, a language with support for graded monads should feature a *sort* of effects or grades, distinct those of from values and types, which can appear in types that represent graded monads. The language should also be *polymorphic* in effects (just like there is parametric polymorphism in types). Orchard and Petriček [OP14] attempt to embed this into the effect-monad Haskell library.

As an example, consider a graded version of the `State` monad called `PermState` s in which the grade states how the state is used: is it only read (R), only written to (W), never used (P) or both read from and written to (RW). Then an expression e of type `PermState Int R String` represents a stateful computation returning `String` in which the `Int`-typed state is only ever read from. Binding e with an expression k of type `String -> PermState Int W Bool` gives an expression $e \gg= k$ of type `PermState Int RW Bool`.

This example already illustrates the main difficulty of working with graded monads in existing programming languages. Using a separate sort for effects (grades) usually means that we actually have to encode them as types. The language requires some support for type-level operators in order to represent the underlying multiplication operator of the grading algebra. This means that an effect-level equational logic is needed for all practical typing: $R \odot W$ is equivalent to RW , $P \odot R$ is equivalent to R and so on. As a consequence, the hypothetical language should be able to infer that the type of the `repeatTick` function, defined as follows:

```
repeatTick n =
  if n <= 0 then return ()
  else { x <- get; put (x + 1); repeatTick (n - 1) }
```

is `State s RW ()`, as $R \odot W \odot R \odot W \odot \dots \odot R \odot W \odot P = RW$. Most programming languages have weak support for such type-level equations. This creates a gap between what existing type inference algorithms are capable of and what we expect from program analysis.

Potential of graded monads

Monads are well understood and widely used in Haskell, yet graded monads have not caught on. I argue that this is mostly due to lack of linguistic support for graded monads in general-purpose languages: grades often need to be (uncomfortably) encoded as types. Graded monads generalise the notion of monads and open up

possibilities of writing more interesting programs, yet a programmer wishing to utilise them faces obstacles (which we will describe in upcoming chapters). They open up the possibility of embedding useful type-and-effect analyses into the type system of the language, effectively turning analyses performed by external tools or verified via testing into type checking and type inference problems.

The goal is to allow the programmer to define their own grading algebras and graded monads, embedding analyses suitable for the task at hand. If the programmer needs to use a new kind of side effect and a new analysis, they should be able to introduce it using concise linguistic constructs that make this possible, without relying on ad-hoc embeddings into the type system of a language (such as Haskell).

1.2 Program properties

Before moving on to the contributions of this thesis, I will briefly explain what is more generally thought of as ‘program analysis’, and how type systems can play a role in reasoning about programs.

Static program analysis

The term ‘program analysis’ is typically used to refer to automatically inferring program properties related to correctness or safety. The result of a program analysis can be used for verifying correctness and for program optimisation.

There are various approaches to program analysis, and we focus on the static approaches, which operate by considering only the source code and reporting analysis outcomes without running the program. Type systems are one way of expressing program properties, with type-and-effect systems as a simple extension described in Section 1.1. In the context of optimising compilers, various control-flow and data-flow analyses were developed and are also examples of program analysis, though mostly used for program transformation (for example, dead code elimination).

A significant framework for program analysis is *abstract interpretation*, introduced by the Cousots [CC77]. This is the notion of program analysis assumed throughout this thesis. In short, we can view program properties through the lens of two spaces: a *concrete space* and *abstract space*, with the abstract space being a simplified, decidable analysis that corresponds to the concrete runtime behaviour of a program. A simple example of abstract interpretation is sign analysis: we can sometimes infer the sign of an arithmetic expression – even if it is not constant – by statically considering only the signs of all the variables and expressions in the program, and relying on facts such as ‘multiplying two negative numbers gives a positive number’ and ‘adding two positive numbers gives a positive number’.

Behavioural properties via type systems: linear types

Research in type systems has long looked at the kinds of properties that can be expressed with a suitable type system. Before exploring the motivations for this thesis, we consider a simple example that shows the significance of linguistic support for reasoning about program properties.

Linear types, based on Girard’s linear logic [GSS92] and later expanded by Lafont [Laf88], are commonly used to reason about resource usage. In the simplest example, a language may classify variables as either linear or unrestricted, where

linear variables must be used exactly once, whereas there is no such constraint for unrestricted variables. Such a type system can be used in languages with explicit memory management and prevent bugs such as freeing the same part of memory twice. For example, the following expressions would not be typeable:

```
(lambda x. lambda y. <free x, free y>) z z
```

Linear types belong to the class of *substructural type systems*, which augment type abstractions constructs with the ability to control the uses of a data structure of operation (number and order). [Pie05, Ch. 1] These include affine types, a generalisation of the notion of linear types. Affine types are supported in some languages through the concept of smart pointers, which have notably existed in C++ since the C++11 standard (`std::unique_ptr`) and which are seen in Rust (where they are called *boxes*).

1.3 Contribution

This thesis explores practical considerations in designing programming languages with support for graded monads and in using them for tasks previously regarded as program analysis. The aim is to overcome the existing difficulties of using graded monads effectively. This thesis makes four main contributions:

Analysis-graded monads I define a class of graded monads that specifically represent program analyses. The interpretation of the term ‘program analysis’ expands on Cousot’s view that these can all be represented as *semilattices* whose elements represent possible outcomes of an analysis. I propose a set of axioms for grading algebras of *analysis-graded monads* guided both by the theoretical underpinnings of the thesis goal and by the practicalities of representing analyses in existing functional programming languages. Analysis-graded monads are expressed both in the context of programming (allowing the programmer to express *higher-order effects*), as well as their mathematical foundations. A related contribution of this thesis is identifying the linguistic features necessary in order to make their use easy for the programmer.

Novel graded monads I identify novel graded monads, discuss their utility for program analysis, their practical implementations in Haskell, and the difficulties they expose. One problem is that of excessive boilerplate which is necessary to set up the examples, which means that any modification of the grading algebra results in many tedious updates. The other is the inadequacy of Haskell to represent effects in any programmer-friendly manner: we are forced to represent all effects as types, which leads to problems due to safety constraints in type-level programming.

Symbols and scoped effects I introduce *symbols* as a way of resolving the issue of excessive boilerplate and the lack of semantic link between values, types and effects.

Composing effects I address the issue of using graded monads that represent more than one side-effect and combine more than one program analysis. We consider two special cases of combining analyses: those addressing orthogonal side-effects and those combining identical effects. This leads to notions of *graded monad transformers* and *grade families*.

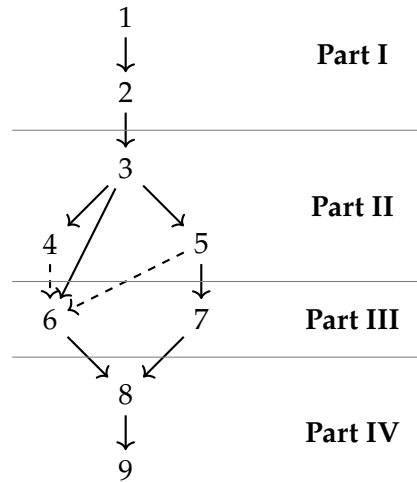


Figure 1.1. Thesis structure and chapter dependencies. A dashed arrow from a to b means that only a part of Chapter b depends on Chapter a .

One of the goals of this thesis is to convince the reader that graded monads are a useful construct whose usability is diminished due to poor facilities in existing mainstream programming languages. I believe that the additions that would make them usable are not difficult to understand and have a solid theoretical foundation.

1.4 Thesis structure

This thesis explores the topic of graded monads in the context of program analysis from a multitude of perspective and identifies several problems (and solutions). The questions considered in the thesis depend on prior results, both from existing work by other authors and other results in this thesis. The thesis is organised into four parts, each with two or three chapters. Each chapter (following this one) depends on some, but not necessary all, previous chapters – for example, Chapter 5 can be understood even if a reader skips Chapter 4. The structure of the thesis is sketched in Figure 1.1.

Part I of the thesis defines graded monads, program analysis and gives other necessary background. **Part II** explores some fundamental aspects of programming with graded monads representing program analysis: the type system, applications, formal semantics, the relationship between type inference and performing an analysis. **Part III** deals with more sophisticated aspects of representing program analyses using graded monads, mainly in the presence of interaction of several effects at the same time. **Part IV** concludes the thesis.

Chapter 2 of this thesis provides the relevant athematical and programming background for the main contributions. This includes semi-lattices, effect systems, monads and graded monads in programming and fundamentals of category theory required for a formal treatment of graded monads. Category theory is most prominently used in Chapter 5 and parts of Chapter 7 and is not essential for understanding most of the contributions of this thesis.

Chapter 3 is both the first chapter with an original contribution and can be seen as an extended background chapter. While graded monads are formally defined in Chapter 2, in this chapter the context of analysis take centre stage. From this point

onward, all graded monads considered in the thesis fall under the umbrella of *analysis-graded monads*. A language is introduced with support for graded monadic types and it is used as the framework for discussing questions about representing analyses throughout the thesis. A significant contribution of this chapter is also the presentation of a zoo of examples of graded monads that are all given in some way novel treatment either in this thesis or in the relevant past publications. In this chapter I also look at practical concerns that arise from implementing these solutions in existing programming languages, which are the main drivers of some of the further chapters.

Chapter 4 considers the matter of automatically inferring the outcome of a program analysis, within the context of a language with support for analysis-graded monads.

Chapter 5 is a short chapter that looks at analysis-graded monads and programming with them from a categorical point of view. The aim is to investigate their structure and state useful structural properties that carry significance in practical programming.

Chapter 6 addresses one of the problems identified in Chapter 3: excessive boilerplate when writing graded monadic code utilising parameterised grading algebras. A graded monad representing stateful computation with multiple components – say variables X, Y, Z – has a long definition that the programmer has to change in a predictable and tedious if a new variable W is introduced. This problem arises in all existing languages in which we can represent graded monads, as the idiomatic monadic approach using monad transformers makes no such rewriting necessary. The proposed solution is the introduction of a kind of symbols, which fill the gap between values and effects. Two languages are considered: ML_G and System G . The former is similar to ML : it has parametric polymorphism in types and effects, as well as decidable type inference. The latter is more akin to System F in that it is a fundamental calculus in which all polymorphism is explicit.

Chapter 7 addresses the question of composing effects and analyses. These are actually two questions: firstly, how can we write programs that feature two different kinds of effects; secondly, how do we merge two analyses that address the same kind of effect? We answer the first question by defining graded monad transformers by proposing a programming language syntax and explaining their expected semantics. The second question is answered using the concept of analysis merging.

Chapter 8 proposes *Granalyse*, a simple programming language that addresses the fundamental limitation of Haskell in writing graded monadic code: lack of facilities for defining effect algebras because there is no *sort* of effects and because type-level operators have restrictive constraints. *Granalyse* introduces the sort of effects and is really two languages in one: the first is the language of effects that supports user-defined grading algebras, while the second language utilises these algebras in a setting that extends that of Haskell.

Chapter 9 outlines interesting avenues for further work and reflects on what was done in the thesis.

Throughout the thesis, but in particular in Chapters 3 and 7, there are description of ways in which some of these ideas can be embedded into Haskell. This is the basis of the *analysis-monad* library, which is the main contribution in the form of code in this thesis. While the main text contains some comments on design decisions, the full source code (with additional explanations) is in the Appendix.

Background

This thesis links several topics: program analysis, type systems, program semantics and category theory. The aim of this chapter is to be a foundation for the contributions of the thesis by summarising the relevant background on these topics, as well as introducing consistent terminology and notation.

Fundamentals of program analysis are given in section 2.1, which include facts about lattice theory and abstract interpretation. Section 2.2 defines the notion of monads and graded monads from the perspective of a programmer. The category-theoretic background of monads and graded monads is given in section 2.3. Finally, section 2.4 outlines work related to and adjacent to the work done in this thesis related to graded monads and effect systems.

2.1 Analysis frameworks

There are various frameworks that aim to unify the notion of program analysis. Some basic notions, such as monotone data-flow frameworks by Kam and Ullman [KU77], represent narrow families of program properties. Others are suitable for a wider array of program analyses. In this thesis, Cousot’s idea of *abstract interpretation* is what we mean when we say ‘program analysis’. Furthermore, as we do not aim to represent all program analyses, but only those that represent various impure notions of computation, the focus is on the intersection of abstract interpretation and effect systems.

2.1.1 Lattice theory

Orders and functions

We start the mathematical background of program analysis by reviewing the notion of a partial order.

Definition 2.1. A pair $(E, \leq)$ (for set E and binary relation $\leq \in E \times E$) is a *pre-order* (or *pre-ordered set*) if the relation $\leq$ is reflexive and transitive. An antisymmetric pre-order is also called a *partial order* (or a *partially-ordered set*, *poset* for short).

In a poset, we sometimes have particularly interesting elements – the least and the greatest ones.

Definition 2.2. A poset $(E, \leq)$ has a *top element* (commonly denoted $\top$) if for all $x \in E$, $x \leq \top$. A poset $(E, \leq)$ has a *bottom element* ($\perp$) if for all $x \in E$, $\perp \leq x$.

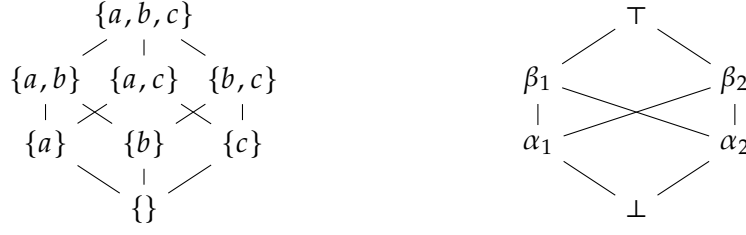


Figure 2.1. Two posets: one of subsets of $\{a, b, c\}$ and another with no particular meaning. The powerset lattice $(\mathcal{P}(\{a, b, c\}), \subseteq)$ on the left has a join $\cup$ and meet $\cap$, whereas the poset on the right has no least upper bound for α_1 and α_2 and no greatest lower bound for β_1 and β_2 .

Commonly we look at two posets $(E_1, \leq_1)$ and $(E_2, \leq_2)$, and consider functions $f : E_1 \rightarrow E_2$ between them. Among these, order-preserving functions occur frequently:

Definition 2.3. A function $f : E_1 \rightarrow E_2$ is said to be *monotonic* with respect to posets $(E_1, \leq_1)$ and $(E_2, \leq_2)$ if, for all $x, y \in E_1$:

$$x \leq_1 y \implies f(x) \leq_2 f(y)$$

Lattices

In program analysis, we often focus on posets with more sophisticated structure. Lattices are a commonly observed structure.

Definition 2.4. A partial order $(E, \leq)$ is a *join-semilattice* when, for any two $x, y \in E$, there exists an element $x \vee y$, which we call their *join*, such that $x \leq x \vee y$, $y \leq x \vee y$ and for all $z \in E$ for which $x \leq z$ and $y \leq z$, we have $x \vee y \leq z$. In other words, the join x and y is the *least upper bound*.

Analogously, a *meet-semilattice* has *meets* $x \wedge y$ for all $x, y \in E$, which are *greatest lower bounds*.

A partial order that is both a join-semilattice and a meet-semilattice is referred to as a *lattice*.

(Semi)lattices can have bottom and top elements, which act as common lower and upper bounds for all pairs of elements. Not all partial orders with top and bottom elements are lattices – Figure 2.1 shows both an example and a counterexample.

In semilattices we commonly consider ascending chains $x_0 \leq x_1 \leq \dots \leq x_n \leq \dots$, both finite and infinite. We can also consider strict chains $x_0 < x_1 < \dots < x_n < \dots$, and if all such strictly ascending chains are finite, we have the notion of *longest ascending chain*.

Definition 2.5. Suppose that in a semilattice all strictly ascending chains $x_0 < x_1 < \dots < x_n < \dots$, are finite and that the maximum number of elements across all these chains exists and equals $h + 1$. Then this semilattice has a finite *height* h .

All infinite-height semilattices have an infinite number of elements, but the converse does not hold – for example, $(\mathbb{N} \cup \{\perp, \top\}, \sqsubseteq)$, where $\perp \sqsubseteq n \sqsubseteq \top$ and $n_1 \not\sqsubseteq n_2$ for $n_1 \neq n_2$, is an infinite lattice with height 2.

It is easy to show that joins and meets (seen as binary operators on E) are commutative and associative. The notion of join and meet is thus easily extended for triples, 4-tuples, and in general all non-empty finite subsets of E – for example, $\bigvee \{a, b, c, d\} = a \vee b \vee c \vee d$. As usual, this does not extend straightforwardly to infinite subsets of E .

Definition 2.6. A poset $(E, \leq)$ is *join-complete* if for all subsets $X \subseteq E$ there exists an element $\bigvee X \in E$ such that for all $x \in X$, $x \leq \bigvee X$, and for all $y \in E$ for which $x \leq y$ for all $x \in X$, $\bigvee X \leq y$.

Analogously define *meet-completeness* ($\bigwedge X$ for $X \subseteq E$).

It is easily shown that join-completeness and meet-completeness are equivalent (given join-completeness, we can define $\bigwedge X = \bigvee \{a \in E \mid \forall x \in X. a \leq x\}$; the converse is analogous), so the term *completeness* is used instead to refer to both. Note that X used in this definition can be empty – in these cases, $\bigvee \emptyset = \perp$ and $\bigwedge \emptyset = \top$, meaning that complete posets have least and greatest elements.

Completeness in this sense implies *chain-(join-)completeness*, meaning that for any ascending chain $x_0 \leq x_1 \leq \dots \leq x_n \leq \dots$ the set $\{x_n \mid n \in \mathbb{N}\}$ has a join (supremum) $\bigvee \{x_n \mid n \in \mathbb{N}\}$. There is also the dual notion of *chain-meet-completeness* which addresses descending chains, but is used less often. Every finite-height poset is also chain-complete.

Definition 2.7. Given two complete posets $(E, \leq)$ and $(E', \sqsubseteq)$, a monotonic function $f : E \rightarrow E'$ is said to be *(join-)continuous* if for any non-empty set $X \subseteq E$:

$$f\left(\bigvee X\right) = \bigsqcup \{f(x) \mid x \in X\}$$

A weaker and more common form of continuity considered in programming language semantics is *chain-(join-)continuity*. This is a property of functions on chain-complete posets. We say a monotonic function $f : E \rightarrow E'$, for chain-complete posets $(E, \leq)$ and $(E', \sqsubseteq)$, is chain-continuous if for any ascending chain $x_0 \leq x_1 \leq \dots \leq x_n \leq \dots$:

$$f\left(\bigvee \{x_n \mid n \in \mathbb{N}\}\right) = \bigsqcup \{f(x_n) \mid n \in \mathbb{N}\}$$

If chain-complete posets $(E, \leq)$ and $(E', \sqsubseteq)$ have finite heights, then all monotonic functions $f : E \rightarrow E'$ are chain-continuous.

One of the simplest, yet most significant, theorems in lattice theory is concerned with fixed points of endofunctions in lattices, and its statement is as follows:

Theorem 2.1 (Knaster–Tarski [Tar55]). Let $(E, \leq)$ be a complete lattice and let $f : E \rightarrow E$ be a monotonic function (with respect to $\leq$). Then the set of fixed points of f in E also forms a complete lattice under $\leq$.

In practice, we are also interested in a more specific statement which gives a method for computing the *least* fixed point for join-continuous functions.

Theorem 2.2. Let $(E, \leq)$ be a chain-complete lattice with bottom element $\perp$ and let $f : E \rightarrow E$ be a chain-continuous function. If the notation f^k refers to the k -fold composition of f (iterated application of f , with $f^0(x) = x$ for all $x \in E$), then the least fixed point of f is given by:

$$\text{fix}(f) = \bigvee \{f^k(\perp) \mid k \in \mathbb{N}\}$$

Many problems in program analysis can be phrased as finding the least fixed point, meaning that a solution can be found by using repeated application of f . The sequence $\perp \leq f(\perp) \leq f(f(\perp)) \leq \dots$ does not necessarily converge, but convergence is guaranteed in finite-height posets.

2.1.2 Type-and-effect systems

Functional languages are frequently extended with constructs for side-effects such as exceptions and references. Accordingly, type systems can be modified so that they capture properties of the side-effect. We briefly outline the idea of *type-and-effect systems*, which serve this purpose.

Effect systems

A simple side-effect we can introduce to a call-by-value functional language is writing to and reading from a finite store of variables, say X, Y, Z . Consider a simple language based on simply-typed lambda calculus with the following syntax for terms:

$$M ::= c \mid x \mid \lambda x : A(M) \mid M N \mid \text{get } V \mid \text{put } V M$$

where V ranges over the mutable variables (the set of which is Vars). Instead of merely considering typing judgements $\Gamma \vdash M : A$ for typing environment Γ , term M and type A , we consider judgements $\Gamma \vdash M : A \ \& \ e$ in which $e \subseteq \bigcup_{v \in \text{Vars}} \{\mathbf{W}_v, \mathbf{R}_v\}$ is an *effect* representing the reading and writing actions possibly performed by the term M .

Since effects can be performed upon term application, types A are also changed: function types are now annotated with a *latent effect*, so we consider types $A \xrightarrow{e} B$, where e represents the effect performed when a term of this type is applied. The type-and-effect system for this language is given in Figure 2.2.

A common addition to programming languages is conditional branching through some kind of **if** term, for example **if** M **then** N_1 **else** N_2 . The effect of a branching term is the union of the effects in the condition and the two branches:¹

$$(\text{IF}) \frac{\Gamma \vdash M : \text{Bool} \ \& \ e \quad \Gamma \vdash N_1 : A \ \& \ f_1 \quad \Gamma \vdash N_2 : A \ \& \ f_2}{\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : A \ \& \ e \cup f_1 \cup f_2}$$

While this is acceptable for A a base type (like **Int**), it is overly restrictive for function types: the effect annotations have to match perfectly. We therefore add a *subeffecting* rule that allows us to *overapproximate* the effect of some term in a typing context:

$$(\text{SUB}) \frac{\Gamma \vdash M : A \xrightarrow{e} B \ \& \ f}{\Gamma \vdash M : A \xrightarrow{e'} B \ \& \ f} \text{ for } e \subseteq e'$$

¹This is clearly only applicable for *may* analyses, in which the desired outcome of the static analysis is a superset of the ‘true’ dynamic outcome.

$$\begin{array}{ll}
(\text{CONST}) \frac{}{\Gamma \vdash c : \mathcal{T}_c \& \{ \}} & (\text{VAR}) \frac{x : A \in \Gamma}{\Gamma \vdash x : A \& \{ \}} \\
(\text{LAMBDA}) \frac{\Gamma, x : A \vdash M : B \& e}{\Gamma \vdash \lambda x : A(M) : A \xrightarrow{e} B \& \{ \}} & (\text{APP}) \frac{\Gamma \vdash M : A \xrightarrow{e} B \& f \quad \Gamma \vdash N : A \& g}{\Gamma \vdash M N : B \& e \cup f \cup g} \\
(\text{GET}) \frac{}{\Gamma \vdash \text{get } v : \text{Int} \& \{ \mathbf{R}_v \}} & (\text{PUT}) \frac{\Gamma \vdash M : \text{Int} \& e}{\Gamma \vdash \text{put } v M : \text{Unit} \& e \cup \{ \mathbf{W}_v \}}
\end{array}$$

Figure 2.2. A type-and-effect system for mutable variables holding integer values.

This rule allows us to, for example, express that a function potentially writing to X and Y when applied is also a function that might write to X , Y and Z when applied. However, this is incomplete: it does not accurately handle subeffecting for higher-order functions. This can be rectified by introducing a subtyping relation, but this introduces further difficulties (which we can ignore for now).

These ideas extend easily to effect systems describing access to communication channels in concurrent programs [NN94] and other safety properties related to concurrent programs [FQ03a, BR01], usage of memory regions [TJ94], safe error-handling [BB07a] and other side-effects. The intended practical application of type-and-effect systems is in performing safe optimisations. This is typically done by also defining an operational semantics for the language, and then showing that the effect of a term is an overapproximation of the side-effects performed by closed terms (terms without free variables, which reduce to values). An example of this idea is given by Tofte and Talpin. [TJ94]

Effect algebras

The above type-and-effect system is in the style of their original formulation, due to Lucassen and Gifford. [LG88] In these type-and-effect systems, the same underlying operator is used for merging effects in conditional branching and in function application – in the case of stateful computation above, that operator is set union. These operators might not always match, and intuitively they should not: running two commands in a sequence fundamentally differs from running exactly one of them (without the knowledge of exactly which one). Furthermore, the operator representing the sequencing of two computations should not be required to be commutative. Tate [Tat13] introduces the term ‘*sequential effect systems*’ to refer to effect systems in which the sequencing operator is may be not commutative.

The most commonly cited examples of sequential effect systems are those concerned with concurrent programs, where order of operations has profound implications on the program safety. For illustration purposes, consider the following example from the CAT imperative language by Flanagan and Qadeer [FQ03b], with concurrent accesses to shared memory. In a concurrent program, we can associate with each term an *atomicity effect*, which represents the extent to which the evalu-

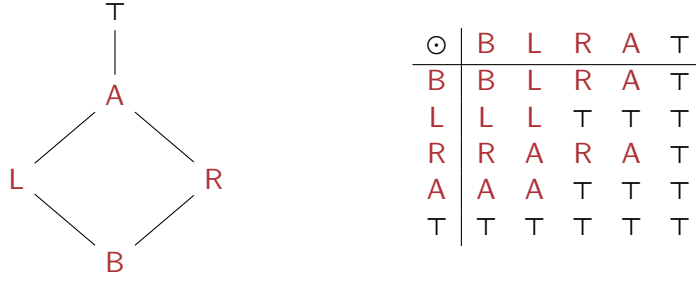


Figure 2.3. The algebra of atomicities by Flanagan and Qadeer [FQ03b]: partial order (lattice) and the sequencing operator.

ation of the term commutes with operations of other threads. The possible effects are:

- **R**: evaluating the term right-commutes with operations of other threads – meaning that, if during execution its action is followed by any action of a different thread, these two actions can be swapped without changing the overall outcome;
- **L**: evaluating the term left-commutes with operations of other threads – if during execution its action is preceded by an action of another thread, these actions can be safely swapped without changing the resulting state;
- **B**: the term is both left-commutative and right-commutative;
- **A**: the term is *atomic*, meaning that it can be seen as a single indivisible action;
- **T**: none of the properties hold.

As before, there is a clear ‘generalisation’ relation $\sqsubseteq$ defined for this set of effects: for $e \in \{L, R\}$ we have $B \sqsubseteq e \sqsubseteq A \sqsubseteq T$. The interesting typing rule is associated with running two computations in a sequence via function application – adapting to our notation:

$$(\text{APP}) \frac{\Gamma \vdash M : A \xrightarrow{g} B \ \& \ e \quad \Gamma \vdash N : A \ \& \ f}{\Gamma \vdash M N : B \ \& \ e \odot f \odot g}$$

Here the binary operator $\odot$ on $\{B, L, R, A, T\}$ is *not* necessarily commutative. The assumed order in which $M N$ is evaluated is call-by-value: first reduce M as much as possible, then reduce N , and finally reduce the application; this order justifies saying that the effect of the application is $e \odot f \odot g$. The definition of this operator is given in the table in Figure 2.3: sequencing a right-commutative action with a left-commutative one gives an action that has no commutativity, but is atomic.

The type-and-effect rule for conditionals also changes:

$$/(\text{IF}) \frac{\Gamma \vdash M : \text{Bool} \ \& \ e \quad \Gamma \vdash N_1 : A \ \& \ f_1 \quad \Gamma \vdash N_2 : A \ \& \ f_2}{\Gamma \vdash \text{if } M \text{ then } N_1 \text{ else } N_2 : A \ \& \ e \odot (f_1 \sqcup f_2)}$$

The intuition behind this rule is that overall action of the conditional branching is the combination of the action of the condition M and the least upper bound of the actions of the two branches.

The operator $\odot$ is associative and it has $\mathbf{B}$ as its unit. It is also *monotonic* with respect to the ordering relation $\sqsubseteq$, meaning that for any $e, e', f, f' \in \{\mathbf{B}, \mathbf{L}, \mathbf{R}, \mathbf{A}, \top\}$ such that $e \sqsubseteq e'$ and $f \sqsubseteq f'$ the following inequality holds:

$$e \odot f \sqsubseteq e' \odot f'$$

Hence, we have a partial order equipped with an order-preserving binary operator $\odot$ that is associative and has a unit. The algebra $(\{\mathbf{B}, \mathbf{L}, \mathbf{R}, \mathbf{A}, \top\}, \sqsubseteq, \odot, \mathbf{B})$ is thus a *partially-ordered monoid* (*pomonoid*).

In general, most treatments of sequential effect systems assume at least the existence of a partial order and a multiplication operator that together give a partially-ordered monoid. From now on in the thesis we refer to the ordering relation as *overapproximation* or *subeffecting* and the monoid multiplication as *sequencing*. Furthermore, we assume that the overapproximation relation gives a join-semilattice structure, where the join matches with the *alternation* operator used to describe the effects of conditional branching. Our choice – partially-ordered monoid with a join-semilattice structure – is not the only one used in literature. The exact assumptions on effect algebras vary by author; we briefly review the various treatments in Section 2.4 (for example, Gordon’s *effect quantales*).

2.1.3 Abstract interpretation

One of the most notable ways program properties are expressed is through the framework of *abstract interpretation*, formalised in a series of publications by Patrick and Radhia Cousot. [CC77, CC92] Abstract interpretation is concerned with verifying program properties that are statically undecidable² by introducing a loss of precision that results in a decidable analysis. A common example concerns integer arithmetic: computing whether the term reduces to an integer value and determining it is undecidable in a Turing-complete language, but we can make guarantees about the sign of the result or the interval within which the result can be found. However, this idea is even more ubiquitous. *Live variable analysis* divides variables into *live* and *dead*, where a variable is called *dead* at a program point if it is guaranteed to be written to before being read in all execution sequences starting at that program point and *live* otherwise. This *semantic* notion of liveness is undecidable, so the notion of liveness used in most compilers and static analysis tools is a *syntactic one*, which instead considers all paths in the *control-flow graph* of a program instead. This framing makes the problem decidable by a simple algorithm, with the cost being the introduction of imprecision into the analysis – however, the returned result is a *safe* (‘conservative’) *overapproximation* of the dynamic behaviour of the program.

Abstract interpretation views a program through the lens of *domains* and relations between them. Program semantics typically operates within the space of a *concrete* domain of a computation, which is undecidable, whereas the analysis outcomes operates within a decidable *abstract* domain. We take the view that concrete domains $\mathcal{C}$ and abstract domains $\mathcal{A}$ all have a complete lattice structure (though this is far from the most general treatment of this topic). An interpretation is seen as a tuple $(\mathcal{D}, \sqsubseteq, \perp, \top, \llbracket \cdot \rrbracket)$ in which $\mathcal{D}$ is a domain of all possible interpretation values, which has a complete lattice structure, as well as an interpretation function.

²By Rice’s theorem [Ric53], any ‘interesting’ property, meaning a non-trivial semantic property, not a property of the source code.

We now adapt the definition of *consistent* abstract interpretation (given by the Cousots) in our presentation of the theory, in the meantime simplifying the notation and the technical details.

Definition 2.8. An abstract interpretation $\bar{\mathbb{I}} = (\mathcal{A}, \bar{\leq}, \perp, \top, \llbracket \cdot \rrbracket_{\mathcal{A}})$ is consistent with a concrete interpretation $\mathbb{I} = (C, \leq, \perp, \top, \llbracket \cdot \rrbracket_C)$ if there exists a pair of monotonic functions $\alpha : C \rightarrow \mathcal{A}$ and $\gamma : \mathcal{A} \rightarrow C$, which we call *abstraction* and *concretisation*, such that:

$$\begin{aligned} \forall x \in C. \quad x &\leq \gamma(\alpha(x)) \\ \forall y \in \mathcal{A}. \quad \alpha(\gamma(y)) &\bar{\leq} y \end{aligned}$$

Furthermore, if $f : C \rightarrow C$ is a map from a concrete input to a concrete output, and in the abstract domain this link is given by a function $\bar{f} : \mathcal{A} \rightarrow \mathcal{A}$, these functions are required to satisfy:

$$\begin{aligned} \forall y \in \mathcal{A}. \quad f(\gamma(y)) &\leq \gamma(\bar{f}(y)) \\ \text{or, equivalently} \\ \forall x \in C. \quad \alpha(f(x)) &\bar{\leq} \bar{f}(\alpha(x)) \end{aligned}$$

The definition is visualised in Figure 2.4. Abstraction $\alpha(x)$ represents the most precise approximation of x , whereas the concretisation $\gamma(y)$ is the most imprecise element of the concrete domain which can be soundly approximated by y . The $\forall x \in C. x \leq \gamma(\alpha(x))$ part of the establishes the idea that $\bar{\mathbb{I}}$ is an approximation: abstraction may introduce some loss of information, so concretising again might result in a larger context, meaning $x \leq \gamma(\alpha(x))$. The $\forall y \in \mathcal{A}. \alpha(\gamma(y)) \bar{\leq} y$ requirement definition requires that concretisation does not introduce loss of information.³ These two conditions mean that α and γ form a *Galois connection* (equivalently, an *adjunction*), which can be equivalently stated concisely as:

$$\forall x \in C, y \in \mathcal{A}. \quad \alpha(x) \bar{\leq} y \iff x \leq \gamma(y)$$

The final part of the definition states that a valid abstraction at least ‘encompasses’ the concrete values.

While the definition implies that the structure is a complete lattice and this has been the treatment thus far, in practice we only need either the join or the meet.

I already mentioned a simple example of finding the sign of an integer-valued arithmetic expression. The concrete domain is the domain of sets of possible values the expression may take – for example, $\{5, 11, 12\}$ is a member of this domain. The abstract domain is much simpler and it is the powerset lattice representing the set of possible signs of the expression: $(\mathcal{P}(\{+, -, 0\}), \cup)$. The abstract value corresponding to the set $\{5, 11, 12\}$ is thus $\{+\}$, meaning that the result is always positive. Similarly, the abstract value corresponding to $\{-3, -1, 0\}$ is $\{-, 0\}$, meaning that the result is either negative or zero. Conversely, the concrete value corresponding to a set of signs has to be as general as possible – if we know that the result is positive, then the range of possible values is the set of all positive integers. This allows us to devise

³It is sometimes (like in Cousots’ 1977 paper [CC77]) seen in the form $\forall y \in \mathcal{A}. \alpha(\gamma(y)) = y$, which implies α is surjective and γ is injective – indeed, many program analysis examples have this stronger property. There are other definitions of abstract interpretation, which go in the opposite direction and relax the Galois connection property. The framework presented in the thesis considers $\alpha(x)$ the most precise abstraction of x (sometimes referred to as ‘the best abstraction’), though in general this might not exist if the abstract domain is not a complete lattice.

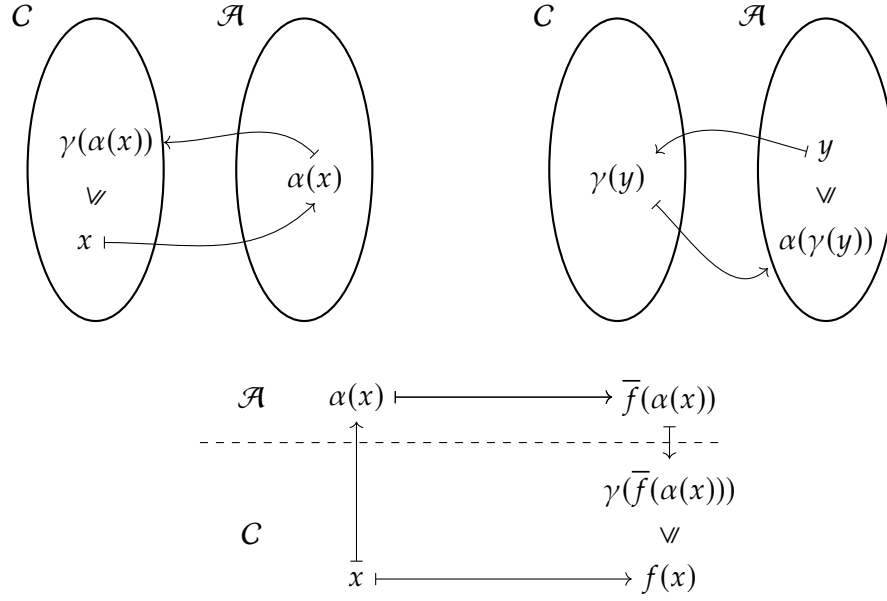


Figure 2.4. Abstract interpretation axioms visualised. The third diagram, connecting abstract and concrete semantics f and $\bar{f}$, is a consequence of the two equivalent properties.

$\alpha(S) = \begin{aligned} & \{0 \mid \text{if } 0 \in S\} \\ & \cup \{+ \mid \text{if } n \in S \text{ for} \\ & \quad \text{some } n > 0\} \\ & \cup \{- \mid \text{if } n \in S \text{ for} \\ & \quad \text{some } n < 0\} \end{aligned}$	$\gamma(\{\})$	$= \{\}$
	$\gamma(\{0\})$	$= \{0\}$
	$\gamma(\{+\})$	$= \{1, 2, 3, \dots\}$
	$\gamma(\{-\})$	$= \{\dots, -3, -2, -1\}$
	$\gamma(\{0, +\})$	$= \{0, 1, 2, 3, \dots\}$
	$\gamma(\{0, -\})$	$= \{\dots, -3, -2, -1, 0\}$
	$\gamma(\{+, -\})$	$= \{\dots, -2, -1, 1, 2, \dots\}$
	$\gamma(\{0, +, -\})$	$= \{\dots, -2, -1, 0, 1, 2, \dots\}$

Figure 2.5. Abstraction and concretisation functions for the sign example.

the abstraction and concretisation functions, fully defined in Figure 2.5. Notice in this example that the top element of the abstract lattice means ‘I don’t know’ – we cannot infer anything about the sign of the final result. On the other hand, the bottom element of the abstract lattice does not mean anything in particular – this outcome only occurs when the set S of concrete values is empty (for example, for a non-terminating program).

Fixed points

Performing an analysis using the abstract interpretation framework in recursive programs or programs with looping constructs requires solving equations of the form $x = f(x)$, for $x \in A$, $f : A \rightarrow A$, $A \subseteq \mathcal{A}$. In other words, x is a fixed point, and among those we are interested in the *most precise* one, meaning the least fixed point: $x = \text{fix}(f)$.

Theorem 2.2 gives an algorithm for finding this solution: starting with $x_0 = \perp$, generate the sequence $x_{k+1} = f(x_k)$ until it becomes stationary. This is an infinite ascending chain of values in A . As we have seen, convergence is not guaranteed if the lattice height is infinite. For lattices that satisfy the *ascending chain property* (meaning that every infinitely ascending chain is ultimately stationary) this can also be a viable option. Otherwise, the attempted solutions are always underestimates – which goes against the idea that an analysis needs to conservatively overapproximate.

Since the aim is to err on the side of safety, viable options for finding an upper bound for $\text{fix}(f)$ are a good approximate solution. Cousots introduce the *widening* and *narrowing* operators [CC77, CC92] as a possible approach.

2.2 Programming with graded monads

We now focus on the other foundational topic of this thesis: *graded monads* and their use in programming. Their overview in this section first recapitulates the use of monads in the context of pure functional programming languages, after which graded monads are framed as their simple generalisation. The focus is on the programming side, as opposed to their use in formal semantics and definitions of theories.

2.2.1 Monads

Programming with monads

In the context of programming, a monad is a type constructor T that, when applied to a type A , results in a type TA representing side-effecting computations whose outcome is of type A . Their significance in pure functional languages was first given by Moggi [Mog89, Mog91]. The aim of monads is to provide a way of neatly structuring side-effecting computations, which includes ways of *sequencing* two side-effecting computations.

Broadly speaking, every monad needs to have two operations: a way of creating a trivial, non-side-effecting computation, which we call *return*; and a way of *binding* a term $M : TA$ with a term $N : A \rightarrow TB$ so that M is ‘run’, and its result is passed as an argument to N , which uses it to produce a new computation whose type is TB . Practically speaking, this means every monad defines a function `return` whose type is $A \rightarrow TA$ for all types A (parametrically polymorphic), and a function `>>=` whose type is $TA \rightarrow (A \rightarrow TB) \rightarrow TB$ for all types A and B , commonly used as an infix operator ($M \gg= N$).

The side-effects that can be represented are various, so we give a brief overview of the most common monads:

Failure The **Maybe** monad represents a way of safely structuring computations that either succeed or fail. This is used as a simplification of exception handling in computations composed of several steps, each of which might fail. The type **Maybe** A is an alias for the sum type $A + \text{Unit}$, with the outcome of type A representing success, and the unit outcome $\langle \rangle$ of type **Unit** representing failure.

Nondeterminism The **List** type constructor is also a monad, and can represent non-deterministic computation. This means that **List** A , the type of lists whose

items are of type A , represents the different outcomes of a computation resulting in a value of type A . The list $[1, 5, 23]$ represents a computation whose possible outcomes are 1, 5 and 23. Sets (which remove duplicates) can represent the same kind of computation.

Reader For any type S , there is a monad Reader_S such that terms of type $\text{Reader}_S A$ represent computations whose outcome has type A and which might read from an additional notable variable whose type is S . The programmer is allowed to use a computation `ask` resulting in the value of the store, meaning that the type of `ask` is $\text{Reader}_S S$. The type $\text{Reader}_S A$ is an alias for $S \rightarrow A$, meaning that the global state is treated as merely an additional argument to a function.

State The idea of reading from a store is extended to full-fledged reading from and writing to a mutable variable – the most common thing identified with the term ‘side-effect’. For any type S , there is a monad State_S such that the type $\text{State}_S A$ represents computations which return a result whose type is A and make use of a store whose type is S . The interaction with the store is performed using `get` and `put` monadic computations, whose types are respectively $\text{State}_S S$ and $S \rightarrow \text{State}_S \text{Unit}$. The $\text{State}_S A$ type is an alias for $S \rightarrow (A \times S)$, which can be seen as a state transformer: given an initial value of the store, return the pair of the result and the most recent value in the store.

Continuations The continuation monad Cont_U for type U is used to represent computations in continuation-passing style (CPS). The type $\text{Cont}_U A$ represents a computation whose outcome has type A , but is expected to be further passed as an argument to a function (called a continuation) which eventually returns a result of type U . This monad can be used to implement complex control flow concepts such as *reified continuations* (call-with-current-continuation). The underlying representation of $\text{Cont}_U A$ is $(A \rightarrow U) \rightarrow U$. The continuation monad is sometimes referred to as ‘the mother of all monads’, as monads whose `bind` and `return` are expressible as purely functional terms (such as all the previously mentioned ones) can be embedded in a call-by-value language with support for composing continuations (which is what the continuation monad provides). [Fil94]

I/O I/O is a kind of computation that breaks purity. In a purely functional language (such as Haskell), monads play two roles: structuring side-effecting computations by internally representing them as pure concepts, as well as providing an interface for programming with genuinely impure concepts such as interaction with the standard input and output, connecting to the Internet or writing to a file. It is not surprising that monads representing ‘true’ I/O require special treatment: the interpreter/REPL and I/O monads conspire to hide a significant amount of truly impure behaviour from the programmer. Such monads are sometimes referred to as being part of the ‘awkward squad’. [PJ02]

Haskell is the most well-known language utilising monads, and it popularised `do` notation for writing monadic code. All monads are instances of the `Monad` type class, whose definition requires implementing a `return` function and a `(>=>)` (bind) operator. As an example, the following computation represents a way of safely computing $\frac{a}{b} + \frac{c}{d} + \frac{e}{f}$, accounting for division by zero:

```
fracArith a b c d e f =
  let safeDiv x y = if y == 0 then Nothing
                    else return (x / y)
  in safeDiv a b >>= \p -> safeDiv c d >>= \q ->
    safeDiv e f >>= \r -> return (p + q + r)
```

A convenient rewrite of this function using `do` shows how monads are similar to programming in an imperative style:

```
fracArith a b c d e f =
  let safeDiv x y = if y == 0 then Nothing
                    else return (x / y)
  in do { p <- safeDiv a b; q <- safeDiv c d;
        r <- safeDiv e f; return (p + q + r) }
```

Monad laws

Not every type constructor T with `bind` and `return` is a monad – there are some additional laws that need to hold. Firstly, returning is meant to produce a computation without underlying side-effects – meaning that it acts as a unit when bound with another computation. The other property of monad is based on the fact that running three computations in a sequence can be bracketed either as $P;(Q;R)$ or $(P;Q);R$, but will give the same outcome.

There are thus two *unit* laws and one *associativity* law, which hold for all $M_1 : TA$, $M_2 : A \rightarrow TB$, $M_3 : B \rightarrow TC$, $N : A$:

$$\begin{aligned} M_1 \gg= \text{return} &= M_1 && (\text{unit-1}) \\ \text{return } N \gg= M_2 &= M_2 N && (\text{unit-2}) \\ (M_1 \gg= M_2) \gg= M_3 &= M_1 \gg= \lambda x. (M_2 x \gg= M_3) && (\text{associativity}) \end{aligned}$$

Monad transformers

The commonly used monads each represent only one kind of side-effect: `State` represents stateful computation, `List` represents nondeterminism and `Maybe` represents failure/error-handling. In large programs, the programmer often needs to keep track of and program with combinations and interactions of such side-effects. The solution given by functional programming languages comes in the form of monad transformers. [Ste94, LHJ95]

A monad transformer R is a map from monads to monads, commonly ‘adding’ another side-effect. For example, the `MaybeT` monad transformer takes a monad T representing side-effecting computations as input and results in a new monad `MaybeT` $[T]$ that represents computations with the same side-effects, but with the added possibility of failure. Each monad transformer R also comes with a *lifting* operation, which serves to translate computations represented via monad T into computations expressed using the monad $R[T]$. In other words, $\text{lift} : TA \rightarrow R[T]A$ is polymorphic in A , but also in T .

Haskell monad transformers are available in the standard libraries, in particular transformers and `mtl` (which stands for *Monad Transformer Library*).⁴ A significant number of monads are merely synonyms for applications of monad transformers to the identity monad (for example, `State s a` is internally implemented as `StateT s Identity a`). All monad transformers are instances of the `MonadTrans` type

⁴<https://hackage.haskell.org/package/mtl>

class. Using these libraries allows us to, for example, combine the failure side-effect represented by `Maybe` and the mutable state side-effect represented by `State s` using the monad `MaybeT (State s)`. For example, the following computation increments the state representing age, returning with an error if it is negative:

```
agePlusOne = do age <- lift get
              if age < 0 then MaybeT (return Nothing)
              else lift (put (age + 1))
```

This example can be somewhat simplified: the lifting can be removed because `get` and `put` are not merely associated with `State s`. In Haskell, the `MonadState s` type class represents all monads with the ability to read from and write to a state of type `s`. Naturally, we expect that applying a state monad transformer to any monad results in a `MonadState s` instance, and this is shown using a single Haskell type class instance definition:

```
instance Monad m => MonadState s (StateT s m) where
  get = StateT (\s -> return (s, s))
  put x = StateT (\_ -> return ((), x))
```

We also expect that applying any monad transformer to the `State s` monad, or more generally to any monad that is an instance of `MonadState s`, gives a monad that is also an instance of `MonadState s`. In `mtl` this frequently results in code bloat: for any monad transformer `t`, a separate instance of the following form is written:

```
instance MonadState m => MonadState s (t m) where ...
```

Digression: Monatron and uniform lifting There are alternative approaches to implementing monad transformers that address the absence of an automated way of writing such code (sometimes referred to as the *non-uniform lifting problem*), for example Jaskelioff's *Monatron* library. [Jas11, Jas09b] The fundamental idea of Monatron is that a suitable representation of the implementations of operations needs to exist in order to allow programmers to both perform the operations and lift them through monad transformers. In Monatron, for each side-effect a type is introduced representing a collection of operations expected of a monad representing this side-effect. From stateful computations we expect getting and putting operations, so we define the type `GetPut s m` as follows:

```
type GetPut s m = (m s, s -> m ())
getX :: GetPut s m -> m s
getX (g, _) = g
putX :: GetPut s m -> s -> m ()
putX (_, p) s = p s
```

In particular, we care about those instances of `GetPut s m` where `m` is a monad, on which we can apply a lifting operation by transforming the collection of operations as follows:

```
liftGetPut :: (Monad m, MonadT t) => GetPut s m -> GetPut s (t m)
liftGetPut (g, p) = (lift g, \s -> lift (p s))
```

for one kind of side-effect only two liftings: one representing lifting the side-effect and one representing adding the desired side-effect. When the `StateT s` monad transformer is applied to a monad `m`, a `GetPut` implementation is provided:

```
getPutStateT :: Monad m => GetPut s (StateT s m)
getPutStateT = (StateT (\s -> return (s, s)),
               \s -> StateT (\_ -> return ((), s)))
```

It suffices to implement the following generic instances:

```
instance MonadT (StateT s) where ...
instance Monad m => Monad (StateT s m) where ...
```

In short, lifting the operations is done uniformly for any transformer t (`liftGetPut`), adding the operations is done uniformly for any monad m , and the preservation of monad properties when `StateT` is applied is also uniform. For N side-effects we therefore write $O(N)$ implementations, as opposed to $O(N^2)$ seen in `mt1`.

2.2.2 Graded monads

We finally get to the key notion of *graded monads* in the context of programming. Despite their name, graded monads are *not* themselves monads – instead, they are a generalisation of monads, and every monad can be seen as a graded monad. In short, a graded monad is defined with a *family* of type constructors T^e , where e ranges over a *grading algebra* (sometimes referred to as an *effect algebra*, by analogy with type-and-effect systems) with a multiplication operator $\odot$ and identity $\mathbf{1}$, with the aim of representing the effect inside the grade and appropriately generalising other monadic operations into a graded setting. This is analogous to the framework introduced for effect systems earlier in this chapter (Section 2.1.2).

Unfortunately, prior work has not settled on a uniform definition or naming convention for graded monads, so I start this section by briefly giving an overview of the alternative names describing the same idea:

‘monads’ Wadler and Thiemann [WT03] gave the first systematic connection between effect systems and programming with monads. They show that effect systems (in particular the effect system of Talpin and Jouvelot used for reasoning about memory regions) can be transposed into corresponding monadic systems. However, the use of the term ‘monad’ is inadequate, as the type system of the target monadic language contains types of the form $T^e A$, where e is a translation of the effect from the source language.

indexed monads Early papers by Orchard on graded monads use the term ‘indexed monad’ to refer to what I refer to as ‘graded monads’ in this thesis. However, ‘indexed monad’ is already used for competing generalisations of monads. In topos theory, there is a notion of $\mathbb{C}$ -indexed functor $T : \mathbb{C} \rightarrow \mathbb{C}$ on cartesian categories $\mathbb{C}$ from which it is possible to define $\mathbb{C}$ -indexed monads [Joh97] – here, a $\mathbb{C}$ -indexed functor is a family of functors $T^X : \mathbb{C}/X \rightarrow \mathbb{C}/X$ satisfying certain coherence properties (not relevant for us here).⁵ In programming terms, there is also McBride’s notion of indexed monads [Con11], provided by Haskell’s `index-core` package and the `IMonad` type class. The term is also sometimes used to refer to what are more commonly called *parameterised monads* [Atk09], which aim to mimic Floyd-Hoare logic assertions used in program verification by adding further pre- and post-condition parameters to the type (more on these in Section 2.4).

parametric effect monads The description and formalisation of parametric effect monads given by Katsumata [Kat14] most closely resembles what is meant by

⁵The notation $\mathbb{C}/X$ denotes a *slice category*: a category whose objects are pairs (A, h) , where A are objects in $\mathbb{C}$ and h are morphisms $A \rightarrow X$, and whose morphisms $f : (A_1, h_1) \rightarrow (A_2, h_2)$ are morphisms in $\mathbb{C} : A_1 \rightarrow A_2$ satisfying $h_1 = h_2 \circ f$.

the term ‘graded monad’ in this thesis and in more recent publications. The original aim was to provide a unified semantics of effect systems.

The term ‘graded monad’ is defined in analogy to the idea of *graded algebra* (not to be confused with ‘grading algebra’), which are algebras M with associative operator $*$ with elements indexed by another algebra I with associative operator $\cdot$ – meaning v_i for $i \in I$ are the elements in M – so that $*$ lifts the application of $\cdot$ from I to M :

$$\forall i, j \in I. \quad v_i * v_j = v_{i \cdot j}$$

The term ‘graded’ has similar meanings in many other contexts in mathematics (for example, graded categories),

We now turn to a formal definition of graded monads in a programming language with a static type system:

Definition 2.9. Consider a pre-ordered monoid $\mathcal{E} = (E, \leq, \odot, 1)$, meaning that $(E, \odot, 1)$ is a monoid and $\mathbb{E} = (E, \leq)$ is a pre-order such that $e \odot f \leq e' \odot f'$ for all $e, e', f, f' \in E$ satisfying $e \leq e', f \leq f'$. For *grading algebra* $\mathcal{E}$ whose elements are called *grades*, a $\mathcal{E}$ -*graded monad* is a tuple $(T, \gg=, \text{return}, \text{sub})$ where:

- for every $e \in \mathcal{E}$, T^e is a type constructor;
- $\gg=$ is a family of binary operators $\gg=_{e,f}$ applied to operands whose types are $T^e A$ and $A \rightarrow T^f B$ for all types A, B , and where the type of the result is $T^{e \odot f} B$ (meaning parametrically polymorphic in A and B).
- return is a function whose type $A \rightarrow T^1 A$ parametrically polymorphic in A ;
- sub is a family of functions $\text{sub}_{e,f}$ whose types are $T^e A \rightarrow T^f A$ (polymorphic in A), for all $e \leq f$.

Together, they satisfy the following four equality axioms:

$$\begin{aligned} M_1 \gg=_{e,1} \text{return} & \equiv M_1 & (\text{unit-1}) \\ \text{return } N \gg=_{1,f} M_2 & \equiv M_2 N & (\text{unit-2}) \\ (M_1 \gg=_{e,f} M_2) \gg=_{e \odot f, g} M_3 & \equiv M_1 \gg=_{e,f \odot g} (\lambda x : A. M_2 x \gg=_{f,g} M_3) & (\text{associativity}) \\ \text{sub}_{e \odot f, e' \odot f'} (M_1 \gg=_{e,f} M_2) & \equiv \text{sub}_{e,e'} M_1 \gg=_{e',f'} (\lambda x : A. \text{sub}_{f,f'} (M_2 x)) & (\text{monotonicity}) \end{aligned}$$

for all types A, B, C and grades $e, f, g, e', f' \in E$ satisfying $e \leq e', f \leq f'$, where the types of the terms are:

$$\begin{array}{ll} M_1 & : T^e A \\ M_3 & : B \rightarrow T^g C \\ M_2 & : A \rightarrow T^f B \\ N & : A \end{array}$$

One of the key insights regarding graded monads is that every grade gives a separate type constructor: for $e \neq f$ and a type A , the types $T^e A$ and $T^f A$ need not match.

Despite the name, graded monads are *not* monads. In fact, the converse is true: every monad can be seen as a graded monad graded by the trivial one-element algebra. In that case, sub is the identity function.



Figure 2.6. The grading algebra associated with **GMaybe**.

Some definitions of graded monads exclude the ordering relation, but we include it because every monoid can be seen as a pre-ordered monoid in which the pre-order is the equality relation. Orchard et al. [OWE20] refer to such structures as *unordered graded monads*. The assumptions of ordering and monotonicity are useful because grades are meant to represent properties of computations.

Example 2.1. We consider a simple graded monad representing the kinds of access capability to a state of type S a computation might have: it is either pure and does not depend on the state (**pure**), only reads from the state (**ro**) or it can both read from and write to the state (**rw**). The ordering relation for these capabilities is **pure** $\leq$ **ro** $\leq$ **rw** and they form a grading algebra with the join $\vee$ as multiplication and **pure** as the identity. The graded monad (**CapState** $_S$, $\gg=$, return, sub) is defined as follows:

$$\begin{aligned}
&\text{CapState}_S^{\text{pure}} A = A, \quad \text{CapState}_S^{\text{ro}} A = S \rightarrow A, \quad \text{CapState}_S^{\text{rw}} A = S \rightarrow (A \times S) \\
&M \gg=_{\text{pure},e} N = N M, \quad M \gg=_{\text{ro},e} N = \lambda s : S. N (M s) \\
&M \gg=_{\text{rw},\text{pure}} N = \lambda s : S. \text{let } \langle v, s' \rangle = M s \text{ in } \langle N v, s' \rangle \\
&M \gg=_{\text{rw},\text{ro}} N = \lambda s : S. \text{let } \langle v, s' \rangle = M s \text{ in } \langle N v s', s' \rangle \\
&M \gg=_{\text{rw},\text{rw}} N = \lambda s : S. \text{let } \langle v, s' \rangle = M s \text{ in } N v s' \\
&\text{return } M = M, \quad \text{sub}_{e,e} M = M, \quad \text{sub}_{\text{pure},\text{ro}} M = \lambda s : S. M \\
&\text{sub}_{\text{pure},\text{rw}} M = \lambda s : S. \langle M, s \rangle, \quad \text{sub}_{\text{ro},\text{rw}} M = \lambda s : S. \langle M s, s \rangle
\end{aligned}$$

Example 2.2 ([SNN94]). It is occasionally possible to statically reason about whether a potentially failing computation is guaranteed to succeed or guaranteed to fail. We thus generalise the **Maybe** monad to **GMaybe**, graded over the algebra $(\{T, F, U\}, \sqsubseteq, \odot, T)$ in which **T** represents computations that cannot fail, **F** represents those which always fail, and **U** means that the answer cannot be inferred ('**U**' stands for 'undecidable'). The grading algebra is described on Figure 2.6.

The graded monadic operations are given as follows:

$$\begin{aligned}
&\text{GMaybe}^T A = A, \quad \text{GMaybe}^F A = \text{Unit}, \quad \text{GMaybe}^U A = A + \text{Unit} \\
&M \gg=_{T,e} N = N M, \quad M \gg=_{F,e} N = \langle \rangle, \quad M \gg=_{U,F} N = \langle \rangle \\
&M \gg=_{U,e} N = \text{case } M \text{ of } (\text{inl } x \Rightarrow (\text{case } N x \text{ of } (\text{inl } y \Rightarrow \text{inl } y \mid \text{inr } \langle \rangle \Rightarrow \text{inr } \langle \rangle))) \\
&\quad \mid \text{inr } \langle \rangle \Rightarrow \langle \rangle \\
&\text{return } M = M, \quad \text{sub}_{e,e} M = M, \quad \text{sub}_{T,U} M = \text{inl } M, \quad \text{sub}_{F,U} M = \text{inr } M
\end{aligned}$$

Programming with graded monads Facilities for practical programming with graded monads are limited in existing programming languages. The two most

significant libraries are in Haskell: `ixmonad`⁶ and `effect-monad`,⁷ by Orchard and Petriček [OP14]. The key challenge is the way grades are represented, as they are a separate sort to that of types and values. The approach taken by these libraries is to encode grades as types through the use of a variety of Haskell extensions, for example `DataKinds` and `TypeFamilies`. The key idea is that grades are represented as types and the key grading algebra operations are represented using *type-level programming* facilities that exist in Haskell, for example type families [KJS10].

2.3 Category-theoretic view

We finally turn to the more fundamental background on graded monads, meaning their more formal treatment in category theory. This endeavour is useful because it allows us to formally explore the semantics of programming languages with effects.

The definitions stated in this section assume knowledge of commonly used concepts in category theory, for example functors, natural transformations, and monoidal categories. Later chapters in this thesis use other ideas, for example colimits. All of the category-theoretic background is summarized in Appendix B.

2.3.1 Monads and graded monads

As noted above, graded monads can be seen as generalising monads, and we elaborate the way in which this holds for categories by first stating the definition of monads.

Definition 2.10. For a category $\mathbb{C}$, a *monad* is a triple (T, μ, η) , where $T : \mathbb{C} \rightarrow \mathbb{C}$ is a functor, $\mu : T \bullet T \Rightarrow T$ (recall $\bullet$ denotes functor composition in this thesis) is a natural transformation (*multiplication*) and $\eta : \text{Id}_{\mathbb{C}} \Rightarrow T$ is a natural transformation (*unit*), satisfying the following laws for all $A \in \mathbb{C}$:

$$\begin{aligned} \mu_A \circ T\mu_A &= \mu_A \circ \mu_{TA} \\ \mu_A \circ T\eta_A &= \text{id}_{TA} \\ \mu_A \circ \eta_{TA} &= \text{id}_{TA} \end{aligned}$$

In other words, the following diagrams commute:

$$\begin{array}{ccc} T \bullet T \bullet T & \xrightarrow{T\mu} & T \bullet T \\ \mu_T \downarrow & & \downarrow \mu \\ T \bullet T & \xrightarrow{\mu} & T \end{array} \qquad \begin{array}{ccc} T & \xrightarrow{T\eta} & T \bullet T \xleftarrow{\eta_T} T \\ & \searrow & \downarrow \mu \nearrow \\ & & T \end{array}$$

A frequent jokey concise definition of a monad is ‘a monoid in the category of endofunctors’. This is technically correct, but requires context. For a category $\mathbb{C}$, there is a monoidal endofunctor category $(\mathbf{Endo}(\mathbb{C}), \bullet, \text{Id}_{\mathbb{C}})$, in which objects are functors $\mathbb{C} \rightarrow \mathbb{C}$ and morphisms are natural transformations between functors. Definition 2.10 precisely matches the definition of a monoid in this category.

This definition is not significant for the purposes of the thesis, but the insight that there is a monoidal category of endofunctors on $\mathbb{C}$ is. Monoids in any monoidal

⁶<https://hackage.haskell.org/package/ixmonad>

⁷<https://hackage.haskell.org/package/effect-monad>

category $(\mathbb{C}, \otimes, I, \alpha, \lambda, \rho)$ can be alternatively phrased in terms of lax monoidal functors from the trivial monoidal category $\mathbb{1}$. To be more precise, a lax monoidal functor from $\mathbb{1}$ (with $*$ as the only object) to $(\mathbf{Endo}(\mathbb{C}), \bullet, \text{Id}_{\mathbb{C}})$ is a triple $(\widehat{T}, \widehat{\mu}, \widehat{\eta})$, where $\widehat{\mu}_*$ and $\widehat{\eta}$ are coherence natural transformations and $\widehat{T}$ is a functor $\mathbb{1} \rightarrow \mathbb{C}$. Then one gets a monad (T, μ, η) defined as $T = \widehat{T}_*$, $\mu = \widehat{\mu}_*$, $\eta = \widehat{\eta}$. This interpretation yields a more general definition of monad graded by monoidal categories.

Definition 2.11. For a monoidal category $\mathcal{M} = (\mathbb{M}, \odot, \mathbf{1}, \alpha, \lambda, \rho)$ and a category $\mathbb{C}$, an $\mathcal{M}$ -graded monad is a lax monoidal functor (T, μ, η) from $\mathcal{M}$ to $(\mathbf{Endo}(\mathbb{C}), \bullet, \text{Id}_{\mathbb{C}})$. In other words:

- for every object $e \in \mathbb{M}$, Te is a functor $\mathbb{C} \rightarrow \mathbb{C}$;
- for every morphism $\sigma \in \mathbb{M}(e, e')$, $T\sigma$ is a natural transformation $Te \Rightarrow Te'$ (corresponding to `sube,e'`);
- for all $e, f \in \mathbb{M}$, $\mu_{e,f}$ is a natural transformation $Te \bullet Tf \Rightarrow T(e \odot f)$ (corresponding to `>>=e,f`);
- η is a natural transformation $\text{Id}_{\mathbb{C}} \Rightarrow T\mathbf{1}$ (corresponding to `return`);

satisfying monotonicity, identity and associativity laws, meaning that the following diagrams for all $e, e', f, f', g \in \mathbb{M}$, with $\sigma : e \rightarrow e', \tau : f \rightarrow f'$:

$$\begin{array}{ccc}
 Te \bullet Tf & \xrightarrow{T\sigma \bullet T\tau} & Te' \bullet Tf' \\
 \downarrow \mu_{e,f} & & \downarrow \mu_{e',f'} \\
 T(e \odot f) & \xrightarrow{T(\sigma \odot \tau)} & T(e' \odot f')
 \end{array}
 \qquad
 \begin{array}{ccc}
 Te & \xrightarrow{T\eta} & Te \bullet T\mathbf{1} \\
 \eta_{Te} \downarrow & & \downarrow \mu_{e,\mathbf{1}} \\
 T\mathbf{1} \bullet Te & & T(e \odot \mathbf{1}) \\
 \mu_{\mathbf{1},e} \downarrow & & \downarrow T\rho_e \\
 T(\mathbf{1} \odot e) & \xrightarrow{T\lambda_e} & Te
 \end{array}$$

$$\begin{array}{ccc}
 Te \bullet Tf \bullet Tg & \xrightarrow{Te\mu_{f,g}} & Te \bullet T(f \odot g) \\
 (\mu_{e,f})_{Tg} \downarrow & & \downarrow \mu_{e,f \odot g} \\
 T(e \odot f) \bullet Tg & & T(e \odot (f \odot g)) \\
 \searrow \mu_{e \odot f, g} & & \swarrow \alpha_{e,f,g} \\
 & T((e \odot f) \odot g) &
 \end{array}$$

This definition of graded monads is very general, but through a few additional assumptions on $\mathcal{M}$ we get the notion relevant for programming. Firstly the unitor and associator natural transformations associated with $\mathcal{M}$, meaning α, λ and ρ , can all be identities, meaning $\mathcal{M}$ is *strict*. This condition introduces equalities to the unit and associativity coherence conditions. Secondly, we can insist that for any two objects $e, e' \in \mathbb{M}$, there is at most one morphism between them – in other words, $\mathbb{M}$ is *thin*. These unique morphisms can then be labelled $e \leq e'$. Strictness and thinness assumptions allows us to consider the underlying monoidal category $\mathcal{M}$ as a pre-ordered monoid: morphisms match the $\leq$ pre-order, $\odot$ is monotonic with respect to $\leq$, and $\mathbf{1}$ acts as the identity for $\odot$.

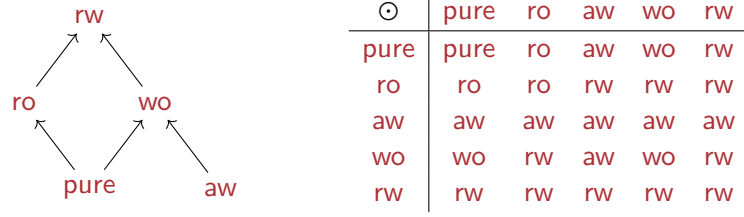


Figure 2.7. The grading algebra associated with the PermState_S graded monad. Arrows are directed to make it clear that the pre-ordered monoid can also be seen as a strict, thin monoidal category.

Example 2.3. Consider the following refinement of Example 2.1, which separates out read and write permissions in stateful computations. The set of permissions is extended with **aw** and **wo**, which respectively represent ‘the computation always writes to the state without reading from it’^a and ‘the computation does not read from the state, but might write to it’. The partial order and the multiplication operator (with identity **pure**) are given in Figure 2.7.

This gives a graded monad we call PermState_S for a cartesian-closed category $\mathbb{C}$ with binary sums and terminal object 1, where S is the object representing the type over which the state ranges. The functor PermState_S is given as follows:

$$\begin{aligned}
 \text{PermState}_S \text{ pure } A &= A & \text{PermState}_S \text{ ro } A &= A^S \\
 \text{PermState}_S \text{ aw } A &= A \times S & \text{PermState}_S \text{ wo } A &= A \times (S + 1) \\
 \text{PermState}_S \text{ rw } A &= (A \times S)^S \\
 (\text{PermState}_S(\text{pure} \leq \text{ro}))_A &= \lambda \pi_1^{A,S} & (\text{PermState}_S(\text{pure} \leq \text{wo}))_A &= \langle \text{id}_A, !_A \rangle \\
 (\text{PermState}_S(\text{aw} \leq \text{wo}))_A &= \text{id}_A \times \iota_1^{S,1} & (\text{PermState}_S(\text{ro} \leq \text{rw}))_A &= \lambda \langle \text{ev}_{A,S}, \pi_2^{A^S, A} \rangle \\
 (\text{PermState}_S(\text{wo} \leq \text{rw}))_A &= \text{id}_{A \times S}
 \end{aligned}$$

with the other parts of the definition being defined via composition, for example:

$$\text{PermState}_S(\text{pure} \leq \text{rw}) = \text{PermState}_S(\text{ro} \leq \text{rw}) \circ \text{PermState}_S(\text{pure} \leq \text{ro})$$

The unit natural transformation is simple: $\eta_A = \text{id}_A$ for all $A \in \mathbb{C}$. The definition of graded multiplication requires defining the natural transformation for all pairs of grades, which number $5^2 = 25$ in total. Instead of stating all twenty-five cases, we give a few simple ones:

$$\begin{aligned}
 (\mu_{\text{pure}, e})_A &= \text{id}_{\text{PermState}_S e A} & (\mu_{e, \text{pure}})_A &= \text{id}_{\text{PermState}_S e A} \\
 (\mu_{\text{aw}, \text{ro}})_A &= \langle \text{ev}_{A,S}, \text{id}_S \rangle & (\mu_{\text{rw}, \text{rw}})_A &= \lambda (\text{ev}_{A \times S, S} \circ \text{ev}_{(A \times S)^S \times S, S})
 \end{aligned}$$

^athis is a ‘must’ property, whereas the previous ones were ‘may’ properties

Digression about the permissions algebra The algebra in Example 2.3 is a somewhat strange, with the main surprise coming from the asymmetry between reads and writes. There is a view that ‘restores’ this symmetry: we can introduce grades representing ‘no read’, ‘no write’, ‘always reads, no writing’ and ‘not yet known’, respectively represented by **nr**, **nw**, **ar** and $\perp$, resulting in a lattice with $\perp$ as the

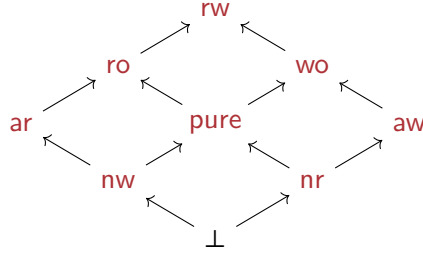


Figure 2.8. Elaboration of the extended grading algebra for permissions mentioned in the digression (multiplication operator not shown).

bottom element of the order, $nr \leq pure$, $nr \leq aw$, $nw \leq pure$, $nw \leq ar$ and $ar \leq ro$ (Figure 2.8). There is no meaningful computational difference and no obvious way of distinguishing between ro and ar , the nr and nw grades as introduced here are observationally identical to $pure$, and $\perp$ is not computationally meaningful (it only means there is no information yet). While a ‘join-semilattice structure with top and bottom’ is relevant later in the thesis (Chapter 4), the algebra over which $PermState_S$ is graded can be seen as the result of quotienting the more sophisticated algebra under ‘observational equality’.

Strength In a practical programming context, monads and graded monads need to be equipped with (*tensorial*) *strength* in order to define the semantics of effectful languages.

Definition 2.12. A functor $T : \mathbb{C} \rightarrow \mathbb{C}$ for a monoidal category $(\mathbb{C}, \otimes, I, \alpha, \lambda, \rho)$ is *tensorially strong* if there exists a natural transformation $t_{A,B} : A \otimes TB \rightarrow T(A \otimes B)$ (natural in two parameters) such that the following diagrams commute for all $A, B, C \in \mathbb{C}$:

$$\begin{array}{ccc}
 (A \otimes B) \otimes TC & \xrightarrow{t_{A \otimes B, C}} & T((A \otimes B) \otimes C) \\
 \alpha_{A, B, TC} \downarrow & & \downarrow T\alpha_{A, B, C} \\
 A \otimes (B \otimes TC) & \xrightarrow{id_A \otimes t_{B, C}} A \otimes T(B \otimes C) \xrightarrow{t_{A, B \otimes C}} & T(A \otimes (B \otimes C))
 \end{array}$$

$$\begin{array}{ccc}
 I \otimes TA & \xrightarrow{t_{I, A}} & T(I \otimes A) \\
 \lambda_{TA} \searrow & & \downarrow T\rho_A \\
 & & TA
 \end{array}$$

Strong monads and strong graded monads are, put concisely, equipped with tensorially strong endofunctors that ‘play well’ with unit and multiplication natural transformations.

Definition 2.13. A *strong monad* on a monoidal category $(\mathbb{C}, \otimes, I, \alpha, \lambda, \rho)$ is a tuple (T, μ, η, t) , where (T, μ, η) is a monad on $\mathbb{C}$, T is tensorially strong with

strength t , and the following two diagrams commute for all $A, B \in \mathbb{C}$:

$$\begin{array}{ccccc}
 A \otimes TTB & \xrightarrow{t_{A,TB}} & T(A \otimes TB) & \xrightarrow{Tt_{A,B}} & TT(A \otimes B) \\
 \text{id}_A \otimes \mu_B \downarrow & & & & \downarrow \mu_{A \otimes B} \\
 A \otimes TB & \xrightarrow{t_{A,B}} & T(A \otimes B) & &
 \end{array}$$

$$\begin{array}{ccc}
 A \otimes B & & \\
 \text{id}_A \otimes \eta_B \downarrow & \searrow \eta_{A \otimes B} & \\
 A \otimes TB & \xrightarrow{t_{A,B}} & T(A \otimes B)
 \end{array}$$

Definition 2.14 ([Kat14]). A *strong $\mathcal{M}$ -graded monad* on a monoidal category $(\mathbb{C}, \otimes, I, \alpha, \lambda, \rho)$ is a tuple (T, μ, η, t) , where (T, μ, η) is an $\mathcal{M}$ -graded monad on $\mathbb{C}$, Te is tensorially strong with strength t_e for all $e \in \mathbb{M}$, and the following diagrams commute for all $e, f \in \mathbb{M}$ and $A, B \in \mathbb{C}$:

$$\begin{array}{ccccc}
 A \otimes Te(TfB) & \xrightarrow{(t_e)_{A,TfB}} & Te(A \otimes TfB) & \xrightarrow{Te(t_f)_{A,B}} & Te(Tf(A \otimes B)) \\
 \text{id}_A \otimes (\mu_{e,f})_B \downarrow & & & & \downarrow (\mu_{e,f})_{A \otimes B} \\
 A \otimes T(e \odot f)B & \xrightarrow{(t_{e \odot f})_{A,B}} & T(e \odot f)(A \otimes B) & &
 \end{array}$$

$$\begin{array}{ccc}
 A \otimes B & & \\
 \text{id}_A \otimes \eta_B \downarrow & \searrow \eta_{A \otimes B} & \\
 A \otimes T1B & \xrightarrow{(t_1)_{A,B}} & T1(A \otimes B)
 \end{array}$$

The settings in which these definitions are used are cartesian closed categories, as they have a monoidal structure with $\otimes = \times$ and $I = 1$. Strength is a necessary requirement for defining the semantics of effectful computations: typically the semantics of $\Gamma \vdash M : TA$ is $\llbracket M \rrbracket : \llbracket \Gamma \rrbracket \rightarrow T\llbracket A \rrbracket$, and strength is a necessary component for defining the semantics of monadic binding.

2.3.2 Categories associated with monads

Prior work by Street [Str72a] defines the category of monads: objects are monads in a category $\mathbb{C}$, whereas morphisms are categorial analogues of the concept of monoid homomorphism.

Definition 2.15. For a category $\mathbb{C}$, the *monad category* $\mathbf{Mnd}(\mathbb{C})$ is:

objects monads (T, μ, η) on $\mathbb{C}$;

morphisms a morphism from (T_1, μ_1, η_1) to (T_2, μ_2, η_2) is a natural transformation $\phi : T_1 \Rightarrow T_2$ coherent with the multiplication and unit natural

transformations, meaning that the following diagrams commute:

$$\begin{array}{ccc}
 T_1 \bullet T_1 & \xrightarrow{\mu_1} & T_1 \\
 \varphi \bullet \varphi \downarrow & & \downarrow \varphi \\
 T_2 \bullet T_2 & \xrightarrow{\mu_2} & T_2
 \end{array}
 \qquad
 \begin{array}{ccc}
 \text{Id}_{\mathbb{C}} & \xrightarrow{\eta_1} & T_1 \\
 & \searrow \eta_2 & \downarrow \varphi \\
 & & T_2
 \end{array}$$

composition natural transformation composition;

identities identity natural transformations $T \Rightarrow T$.

For a particular monad T we also typically define the *Kleisli* and *Eilenberg-Moore* categories. The ideas from the former are in particular used in defining the semantics of binding (especially the Haskell-like $\gg=$ operator).

Definition 2.16. For a monad (T, μ, η) on $\mathbb{C}$, its Kleisli category $\mathbb{C}_T$ is defined as follows:

objects same as in $\mathbb{C}$;

morphisms every morphism $f : A \rightarrow TB$ in $\mathbb{C}$ can also be regarded as a morphism $A \rightarrow B$ in $\mathbb{C}_T$;

composition composition $\circ_T$ for $f : A \rightarrow B$, $g : B \rightarrow C$ in $\mathbb{C}_T$ is defined as $g \circ_T f = \mu_C \circ Tg \circ f$;

identities the identity morphism for all $A \in \mathbb{C}_T$ is $\text{id}_A = \eta_A$.

All of these ideas extend to a graded setting, with the main contributions given by Fujii and Katsumata. [FKM16] These ideas are further explored in Chapter 7.

2.3.3 Monad transformers as monoid transformers

The most significant formal treatment of monad transformers is given by Jaskelioff in his thesis [Jas09a] and his work with Moggi [JM10a]. This treatment considers monad transformers as a special instance of *transformers* and in particular *monoid transformers*.

Definition 2.17. Given a category $\mathbb{B}$ and some subcategory $\mathbb{A}$ of $\mathbb{B}$ with inclusion functor $\text{Inj} : \mathbb{A} \rightarrow \mathbb{B}$, a *transformer* (R, lift^R) on $\mathbb{A}$ is a pair of a functor $R : \mathbb{A} \rightarrow \mathbb{B}$ and natural transformation $\text{lift}^R : \text{Inj} \Rightarrow R$.

The subcategory $\mathbb{A}$ in question is usually either $\mathbb{B}$ itself or its discretisation $|\mathbb{B}|$, with all the objects of $\mathbb{B}$, but only identity morphisms. A *monoid transformer* on a monoidal category $\mathcal{C} = (\mathbb{C}, \otimes, I, \alpha, \lambda, \rho)$ is then simply a transformer on $\mathbf{Mon}(\mathcal{C})$, the category of monoids on the $\mathcal{C}$. Since monads are monoids in the monoidal categories of endofunctors, monad transformers are transformers on $\mathbf{Mon}(\mathbf{Endo}(\mathbb{C})) = \mathbf{Mnd}(\mathbb{C})$.

Jaskelioff and Moggi [JM10a] introduce a hierarchy of monoid transformers, in strictly increasing order of underlying assumptions. The simplest monoid transformers assume $\mathbb{A} = |\mathbf{Mon}(\mathcal{C})|$, $\mathbb{B} = \mathbf{Mon}(\mathcal{C})$, meaning that R' in Definition 2.17 is only an object mapping, there is no functoriality on the morphisms in $\mathbf{Mon}(\mathcal{C})$.

The additional functoriality assumption, meaning $\mathbb{A} = \mathbb{B} = \mathbf{Mon}(\mathcal{C})$, yields *covariant* monoid transformers. A *functorial* monoid transformer is a covariant monoid transformer (R, lift^R) on $\mathbf{Mon}(\mathcal{C})$ along with an underlying transformer on $\mathbb{C}$ also denoted (R, lift^R) , as well as a coherent inclusion functor $U : \mathbf{Mon}(\mathcal{C}) \rightarrow \mathbb{C}$, meaning that the following diagram (in **Cat**) commutes:

$$\begin{array}{ccc}
 \mathbf{Mon}(\mathcal{C}) & \begin{array}{c} \xrightarrow{\text{Inj}} \\ \Downarrow \text{lift}^R \\ \xrightarrow{R} \end{array} & \mathbf{Mon}(\mathcal{C}) \\
 \downarrow U & & \downarrow U \\
 \mathbb{C} & \begin{array}{c} \xrightarrow{\text{Inj}} \\ \Downarrow \text{lift}^R \\ \xrightarrow{R} \end{array} & \mathbb{C}
 \end{array}$$

The maximum requirement gives *monoidal* monoid transformers, which are based on the idea of monoidal extensions (not defined here). This hierarchy can be shown to be proper (strict).

A significant part of the theory of monad transformers deals with the way operations are lifted: how is reading from and writing to a state defined when a non-stateful monad transformer is applied to a state monad? Jaskelioff's thesis is the go-to reference for such questions (in which the aforementioned monad transformer hierarchy gives some automatic answers) and we do not discuss them further.

2.4 Related work

Previous sections in this chapter considered the mathematical foundations and topics that are directly used in the main contribution chapters of the thesis. This section addresses the related work and discusses alternative approaches to some of the introduced topics. The focus is first on effects (Section 2.4.1), both the type-and-effect systems side of the discussion and the various alternative constructs for programming with effects. Next, I briefly consider a topic dual to monads and effects: comonads and coeffects (Section 2.4.2). Using a type system with support for grades, one of the contributions in this thesis, has been attempted by other languages, and this is discussed in Section 2.4.2. Finally, I briefly consider monad generalisations other than graded monads and their relation to program properties (Section 2.4.3).

2.4.1 Effects

Effect algebras

The treatment of effect algebras in Section 2.1.2 can be seen as placing assumptions on what is required of grading algebras. The least possible requirement – a pre-ordered monoid structure, where multiplication represents sequencing – is in line with the definition of $\mathcal{M}$ -graded monads when $\mathcal{M}$ is a strict and thin monoidal category. However, the use of type-and-effect systems in practice places requirements that aim to ensure expressiveness and decidability of the resulting analysis. The simplest additional requirement hinted in Section 2.1.2 is that the effect algebra should have a join-semilattice structure (because of conditional branching). The traditional Lucasses-Gifford-style [LG88] effect systems stick to effect algebras with

a complete lattice structure in which multiplication is the join – this is the definition given by Nielson, Nielson and Hankin in their influential textbook. [NRNH99, p. 291–300] We explore more recent and more general assumptions made by other authors.

Tate [Tat13] introduces the term ‘sequential effects’ to refer to non-commutative effects, whose semantics are expressed using the notion of *productors* with respect to an *effectoid*. Tate’s definition of a productor is almost equivalent to that of a graded monad and the *effectoid* corresponds to a grading algebra. The main difference lies in the fact that an effectoid is not an algebra because sequencing is represented by a partial function $E \times E \rightarrow E$ – it is not a binary operator. This is useful when a type system is meant to reject unsafe programs, and the lack of safety follows from sequencing two computations deemed ‘incompatible’. Such partial operators can be rephrased as binary operators on a set extended with an additional element representing failure or unsafety.

More recently, Gordon has paid closer attention to control-flow operations and their interaction with effects [Gor17, Gor18, Gor21]. These operations include conditional branching, loops, exceptions, call-with-current-continuation and so on. The resulting structure of the effect algebra is an *effect quantale*. In short, an effect quantale is a monoid along with a join-semilattice with a nilpotent (absorptive) top $\top$ ($\forall e \in E. \quad e \odot \top = \top \odot e = \top$), in which the sequencing operator distributes over the join on both sides:

$$\forall a, b, c \in E. \quad a \odot (b \vee c) = (a \odot b) \vee (a \odot c) \quad \text{and} \quad (a \vee b) \odot c = (a \odot c) \vee (b \odot c)$$

For modelling loops, there is need to consider *iterable* effect quantales, which feature an *iteration* (elsewhere *closure*) operator similar to ones seen in Kleene algebras [Koz90], modelling unbounded application of the sequencing operator: e^* can be seen as the limit of $e \odot e \odot e \odot \dots$

Semantics of effects Defining a formal semantics for effects has long been a research topic, starting with Moggi’s *computational λ -calculus* λ_c [Mog89], with a monadic categorical semantics. More recently, the semantics of Lucassen-Gifford-style type-and-effect systems was explored in great detail by Ohad Kammar, particularly in his PhD thesis [Kam14]. Orchard et al. [OPM14] define the semantics of an effectful language in a more general setting, with a monotonic sequencing operator, via graded monads. In parallel, Petriček et al. [PS11] propose *joinads* to model effectful computation in concurrent and reactive settings, which are extended to a graded setting by Mycroft et al. [MOP16], with grades being representations of effects.

Programming with effects

Extensions of λ -calculus with effectful computations and programming with (graded) monads are not the only ways of representing impure computations. Furthermore, the insight behind effect systems enables further programming paradigms, to which we briefly pay attention now. The focus on *algebraic effects and handlers* and *delimited continuations* is significant because of the fact these constructs can be shown to be equivalent to graded monads in terms of expressiveness.

Algebraic effects and handlers The fundamental intuition behind [PP09, PP13] is that exception-handling syntax is extended to encompass all computational effects,

for example I/O and mutable stores. We talk about *algebraic* effects because it is assumed that they arise from a finite set of operations (raising exceptions, getting/-setting the state, reading/printing), and their handlers specify how encountering such an effect during program execution needs to be processed. This allows us to capture more sophisticated control flow that encompasses backtracking, exceptions, coroutines, as well as logic and probabilistic programming.

There are several programming languages with support for algebraic effects. Eff [BP12] is a language with OCaml-like syntax with first-class effects – that is, effects can be programmer defined. Support for algebraic effects in handlers exists in languages such as Koka, Frank, and F*. There are also language extensions which add support for algebraic effects and handlers in Haskell, OCaml, and Scala.

I now briefly explain how algebraic effects and their handlers are used through a simple example using Eff-like syntax. All effect handlers have a continuation parameter representing the *context* in which an effect is performed, and the programmer is allowed to *continue* the execution of the effectful computation. Exception handlers are a special case of effect handlers. They can be defined as an Exn effect:

```
effect Exn : string -> unit
```

and we assume that the language has support for strings, with a ++ operator used for string concatenation. Furthermore we assume the language has I/O operations, with a dedicated print function for writing to standard output. The following function hello can then be written:

```
let hi name = if name = "" then perform (Exn "no_name_given")
               else print ("Hi,_" ++ name ++ "!" )
```

In order to achieve the stack unwinding behaviour of exceptions, we just do not use the continuation (exncont) in the handler at all. For example, the expression:

```
handle hi "Dmitrij"; hi "Nathanael"; hi ""; hi "Andrej"
with Exn m exncont -> print ("Exception:_" ++ m)
```

prints:

```
Hi, Dmitrij!
Hi, Nathanael!
Exception: no name given
```

In short, algebraic effects and handlers can be seen as a generalisation of exception handling to other kinds of effects, along with the option to resume a computation.

Delimited continuations connection *Delimited continuations* [Fel88, DF90] are one of the ways of programming with continuations in order to represent control-flow not typically seen in pure functional programs. They extend constructs such as call/cc, and the primitives used to program with them can vary. Commonly commands such as shift and reset are used, which are similar to the homonymous concepts in parsing.

Forster et al. [FKLP17] show that delimited continuations, monads and algebraic effects and handlers all have the same expressibility. This can be seen by constructing translations (effectively, how one can simulate the other). Some of the translations given in their work do not preserve typeability. However, this typeability issue disappears by changing the type system to introduce features such as parametric polymorphism.

2.4.2 Comonads and coeffects

Dual to effects, the notion of *coeffects* is used to reason about computations which *rely* on the environment in which they are run (as opposed to affecting it, like with effects). First suggested by Petriček et al. [POM13], coeffects are used to model concepts such as implicit function parameters, bounded use of variables (similarly to linear type systems), capability-based secure data access and data-flow programming in the style of Lucid [WA85]. Dually to judgements of the form $\Gamma \vdash M : A \ \& \ F$, coeffects are stated as assumptions along with the context: $\Gamma @ R \vdash M : A$, with R a coeffect. Petriček in his PhD thesis [Pet17] identifies two broad classes of coeffects: *flat* and *structural*. Flat coeffects are used to represent an assumption on the entire environment (for example, the total number of times any of the variables can be used), whereas structural coeffects [POM14] are vectors whose length matches that of the typing environment and represents assumptions associated with each of the variables (for example, an upper bound on the number of times the computation may read them).

The semantics of type-and-coeffect systems is described by *comonads* and *graded comonads* [Orc14, Ch. 5]. Categorically, comonads D on $\mathbb{C}$ are a dual notion to that of monads: instead of a unit $\eta : \text{Id}_{\mathbb{C}} \Rightarrow T$ and multiplication $\mu : T \bullet T \Rightarrow T$, comonads have a *counit* $\varepsilon : D \rightarrow \text{Id}_{\mathbb{C}}$ and *duplication* (or *comultiplication*) $\delta : D \Rightarrow D \bullet D$.

Comonads can be programmed with in a similar way as monads, and the concept predates coeffects. They were first used to describe the semantics of a contextual λ -calculus [UV08] and data-flow programming [UV05] by Uustalu and Vene. Furthermore, Orchard [OM12] introduced *codo* notation, similar to Haskell’s *do*, allowing the programmer to write contextually-dependent computations in a more convenient style. Orchard further demonstrated other applications of comonads in practical programming, for example in image processing. An overview of these ideas can be found in his PhD thesis [Orc14, Ch. 7].

Effects and coeffects can be combined in a type system that account for both ways the environment and a computation may interact with each other – as was done by Gaboardi et al. [GKO⁺16]. This is done by using effect-graded monads and coeffect-graded comonads and the system is capable of reasoning about computations that raise exceptions and have resource requirements. The key insight is that interaction between effects and coeffect is done using a *graded distributive law*: reasoning about resource usage (coeffectful) and exceptions (effectful) needs to account for the fact that, if a computation is known to definitely cause an error, the program flow is interrupted.

Program reasoning

An important use of graded structures (monads, comonads) and effect/coeffect systems is in reasoning about resource usage: whether a file is opened prior to being read from, how many times a particular mutable variable is written to, whether a thread consumes exactly one item from a shared buffer. The most straightforward example of a type system that enables that kind of reasoning is that of Lafont’s linear types [Laf88], which is a substructural type system in which variables are marked as either linear or unrestricted and in which linear variables must be referred to exactly once in the scope within which they are defined. A straightforward generalisation is the notion of *bounded linear types* [GS14, GSS92, DLH10], in which variables can be treated as resources to be consumed, with the exact usage discipline described by

a resource semiring. The categorical semantics of linear and bounded linear type systems can be expressed comonadically [Bie95], which suggests that reasoning about resources can be done using effects and coeffects.

Granule (formerly Gram) [OL17, OLEI19] is a programming language designed around *quantitative program reasoning* by unifying graded comonadic and graded monadic types into an umbrella notion of *graded modal types*. Granule’s static type system is capable of reasoning about properties related to data usage. As these properties can be sophisticated and possibly undecidable, Granule utilises the Z3 theorem prover to partially fill the gap in the type-checking algorithm. This way, the programmer is capable of – for example – representing type-safe file handling, without the need for sophisticated type-level programming or dependent types.

2.4.3 Related monad generalisations

The focus in this thesis is on graded monads and their use in representing program analysis. However, this is not the only generalisation of monads that is useful for representing properties of impure computation. We briefly review a few useful generalisations.

Parameterised monads Atkey [Atk09] introduced a generalisation of monads capable of modelling judgements of the form $\Gamma; P \vdash M : A; Q$, where the traditional typing judgement $\Gamma \vdash M : A$ is extended with P and Q , which are the initial and final predicates on states required and produced by a computation M . A parameterised monad is thus constructs a type out of a *precondition*, *postcondition* and a type. Reasoning about program properties through precondition-postcondition assertion pairs is inspired by axiomatic approaches to semantics (such as Floyd-Hoare logic). Categorically, a parameterised monad in a category $\mathbb{C}$ relies on a parameterisation category $\mathbb{S}$ whose arrows are analogous to implications between assertions, so the underlying functor has type $T : \mathbb{S}^{\text{op}} \times \mathbb{S} \times \mathbb{C} \rightarrow \mathbb{C}$. Parameterised monads are capable of representing concepts such as delimited continuations and model properties such as the range of values of a global state.

Category-graded monads Orchard et al. [OWE20] unify the notions of parameterised and graded monads using the terminology of 2-categories, as both can be seen as instances of *2-category-graded monads with a generalised unit*. This is done in several steps using the ‘oidification’ approach to generalising definitions in category theory. The strict monoidal category $\mathbf{Endo}(\mathbb{C})$ is equivalent to a one-object 2-category $\mathbf{Endo}_2(\mathbb{C})$, whose morphisms are endofunctors and 2-morphisms are natural transformations. Then, for a 2-category $\mathbb{I}$, a 2-category-graded monad is a lax 2-functor $\mathbf{T} : \mathbb{I}^{\text{op}} \rightarrow \mathbf{Endo}_2(\mathbb{C})$. This concept can be shown to straightforwardly generalise $\mathcal{M}$ -graded monads, for a monoidal category $\mathcal{M}$. This is not quite enough to subsume the notion of parameterised monads, and the further technical details can be found in the above-cited paper.

Floyd-Hoare assertions A further connection between monads and program verification comes in the form of *Hoare monads* [Swi09] and *Dijkstra monads*. [SWS⁺13, MAA⁺19] Hoare monads came first as a way of adding Floyd-Hoare-style precondition and postcondition assertions to stateful computations in a dependently-typed setting. They were inspired by Hoare Type Theory [NMB06], used for statically

enforcing correct use of side effects. Dijkstra monads go a step further: they are capable of verifying programs with *arbitrary* monadic side effects, not just states. The main idea of Dijkstra monads comes from the concept of *weakest preconditions*: keeping track of the postcondition means the most general matching precondition can also be derived.

Polymonads Polymonads [HBG⁺14] generalise the familiar type of monadic bind, $\forall \alpha, \beta (T\alpha \rightarrow (\alpha \rightarrow T\beta) \rightarrow T\beta)$, to a more general type $\forall \alpha, \beta (T_1\alpha \rightarrow (\alpha \rightarrow T_2\beta) \rightarrow T_3\beta)$ for a suitable triple $(T_1, T_2) \triangleright T_3$, allowing the programmer to sequence side-effecting computations representing different side-effects. Polymonads turn out to be exactly as expressive as Tate’s productoids [Tat13], used for ‘producer’ effect systems (introduced in the same paper as the term ‘sequential effect system’). Polymonads can also express Atkey’s parameterised monads.

Part II

Analysis-graded monads

In this thesis, graded monads are used as a setting for representing program properties through the type system of a language. This leads to the notion of *analysis-graded monads*, in which grades represent outcomes of program analyses. The term ‘program analysis’ is taken to mean ‘abstract interpretation’. Thus analysis-graded monads assume a particular structure of the grading algebra.

Chapter 3 gives a definition of analysis-graded monads in a programming context and provides novel examples of program analyses represented using analysis-graded monads. While the chapter contains novel results, it can also be read as an extended background chapter with definitions used in the remainder of the thesis. In Chapter 4 we take the next step and explore how to perform program analysis by way of *grade inference*, reducing the question of program analysis to that of typing. Finally, in Chapter 5 we look at analysis-graded monads from a category-theoretic perspective, allowing us to state and prove some general properties and useful results.

Program analysis through grades

The aim of this thesis is to bridge the gap between the theory of graded monads and their practical use in analysing side-effecting computations. Simple examples of graded monads were given in Chapter 2, mostly based on prior work. The next step is to explore the contexts in which graded monads represent program analyses, in which case we refer to them as *analysis-graded monads*.

The main motivation for research into analysis-graded monads in particular is to see the ways in which grades can be interpreted in a manner useful for a programmer and tied to a program analysis. Precisely defining analysis-graded monads allows us to state some general results that hold for all of them and to pose interesting questions – for example, how outcomes of multiple analyses can be represented with a single grade.

The goals of this chapter are to show how various program analyses can be thought of as analysis-graded monads and to precisely define them.

Chapter structure I first give a language-oriented definition of analysis-graded monads before focussing on notable concrete examples. A graded monadic metalanguage (called *Source*) is used for discussing this question.

Section 3.1 defines a framework for working with analysis-graded monads using a simple graded monadic metalanguage. This section also illustrates how analysis-graded monads can be embedded into Haskell. In Section 3.2 I explore graded monads that address possibly failing computations and exceptions, meaning graded versions of frequently used *Maybe* and *Except* monads. While using graded monads in these contexts is not a novel idea, I argue that the approach taken in this thesis improves upon previous attempts at representing these side-effecting phenomena. Next, in Section 3.3 I explore graded monads concerned with state, including *PermState*_s mentioned in Chapter 2. The next application is that of analysing concurrent functional programs; several approaches aiming to guarantee safety are described in Section 3.4.

The examples show there are fundamental challenges associated with performing program analysis using graded monads, which are discussed in Section 3.5.

Relevant publications The applications of analysis-graded monads given in this chapter draw upon work I co-authored during my PhD studies, which includes the following publications:

- Andrej Ivašković, Alan Mycroft, Dominic Orchard, *Data-flow analysis as effects and graded monads* (International Conference on Formal Structures for Computation and Deduction 2020) [IMO20] – this publication is the foundation for the work explored in Section 3.3.2 and considers a more general setting of all monotone data-flow frameworks, as opposed to just live variable analysis;

- Andrej Ivašković, Alan Mycroft, *A graded monad for deadlock-free concurrency* (Haskell Symposium 2020) [IM20] – this paper gives a practical Haskell implementation of the simple graded monad for deadlock-free locking in Section 3.4.

3.1 An analysis-graded monadic framework

Programming with monads and graded monads requires representing computations as values whose types are TA or $T^e A$ for some type constructor T and grade e . For example, the **Maybe** monad uses an underlying sum, so the type **Maybe**(A) has $A + \mathbf{Unit}$ as its underlying structure. In practical programming there is usually a data constructor that ‘wraps’ such an underlying structure into a value whose type is (graded) monadic. The (graded) monadic type can be thought of as abstracting over the representation of the computation.

In this section, I define analysis-graded monads in the context of a simple language called *Source*, which is a call-by-name language with graded monadic types. This language is the main theoretical framework used for discussing examples of analysis-graded monads, and as such its type system incorporates judgements related to the grading algebras used by its graded monads. Defining the operational semantics for graded monadic operations (such as binding) depends on the underlying graded monad, which is why the operational semantics is defined in terms of a *target language* *Target* without support for graded monads, but all graded monadic types and computations are replaced using their underlying representation (unwrapped in the same way as the type **Maybe**(A) represents $A + \mathbf{Unit}$). This grade-driven translation from *Source* to *Target* represents a way of reifying these computations.

The *Target* language is sufficient to explain the translation of most examples explored in this chapter. A missing feature is a way of performing I/O, such as writing to an output stream or performing parallel computations. The role of graded monads we consider in the *Target* language is not capturing that of I/O, but that of *structuring computation*. Reifying monadic I/O is known to be technically difficult [PJ02].

We first define the syntax and semantics of *Target* in Section 3.1.1. We then give a definition of *Source* language in Section 3.1.2, along with the definition of analysis-graded monads and their *computational grading algebras*. Finally, we pay attention to the question of representing analysis-graded monads in Haskell. The *Source* and *Target* definitions and translations are mostly mechanical and not novel, but the framework and the examples are.

3.1.1 Defining *Target*

The *Target* language is defined to be as simple as possible: it is based on simply-typed λ -calculus extended with constructors and eliminators for binary sums and finite products. We opt for *call-by-name* reduction semantics for the sake of convenience: programming with (graded) monads is typically done in pure functional programming language such as Haskell, which usually have call-by-name (or similar) reduction semantics.

The syntax, type system and reduction semantics are given in Figure 3.1.

The terms (M, N) of this language range over constants, integer arithmetic, variables (x) , function abstractions and applications, as well as introduction and

Syntax:

$$\begin{aligned}
 \text{(terms)} \quad M, N &::= n \mid \langle \rangle \mid M \text{ op } N \mid x \mid \text{fn } x : A \Rightarrow M \mid M N \\
 &\quad \mid \langle M_1, \dots, M_n \rangle \mid \text{proj}_i M \quad \text{for } i, n \in \mathbb{N} \\
 &\quad \mid \text{inl } M : A + B \mid \text{inr } M : A + B \\
 &\quad \mid \text{case } M \text{ of } (\text{inl } x_1 \Rightarrow N_1 \mid \text{inr } x_2 \Rightarrow N_2) \\
 \text{(values)} \quad V &::= n \mid \langle \rangle \mid x \mid \text{fn } x : A \Rightarrow M \\
 &\quad \mid \langle M_1, \dots, M_n \rangle \\
 &\quad \mid \text{inl } M : A + B \mid \text{inr } N : A + B \\
 \text{(types)} \quad A, B &::= \text{Int} \mid \text{Unit} \mid A \rightarrow B \mid A_1 \times \dots \times A_n \mid A + B
 \end{aligned}$$

Evaluation contexts:

$$\begin{aligned}
 C &::= V \text{ op } _ \mid _ \text{ op } M \mid _ M \mid \langle M_1, \dots, M_{k-1}, _, M_{k+1}, \dots, M_n \rangle \\
 &\quad \mid \text{case } _ \text{ of } (\text{inl } x_1 \Rightarrow N_1 \mid \text{inr } x_2 \Rightarrow N_2)
 \end{aligned}$$

Operational semantics:

$$\begin{aligned}
 \text{(ARITH)} \quad n_1 \text{ op } n_2 &\rightsquigarrow \llbracket \text{op} \rrbracket(n_1, n_2) \\
 \text{(BETA)} \quad (\text{fn } x : A \Rightarrow M) N &\rightsquigarrow M\{N/X\} \\
 \text{(PROJ)} \quad \text{proj}_i \langle M_1, \dots, M_n \rangle &\rightsquigarrow M_i \quad \text{for } 1 \leq i \leq n \\
 \text{(CASE-L)} \quad \text{case } (\text{inl } M : A + B) \text{ of } &(\text{inl } x_1 \Rightarrow N_1 \mid \text{inr } x_2 \Rightarrow N_2) \rightsquigarrow N_1\{M/x_1\} \\
 \text{(CASE-R)} \quad \text{case } (\text{inr } M : A + B) \text{ of } &(\text{inl } x_1 \Rightarrow N_1 \mid \text{inr } x_2 \Rightarrow N_2) \rightsquigarrow N_2\{M/x_2\} \\
 \text{(STRUCT)} \quad \frac{M \rightsquigarrow N}{C[M] \rightsquigarrow C[N]}
 \end{aligned}$$

Typing judgements:

$$\begin{aligned}
 \text{(INT)} \quad \frac{}{\Gamma \vdash n : \text{Int}} \quad \text{(UNIT)} \quad \frac{}{\Gamma \vdash \langle \rangle : \text{Unit}} \\
 \text{(OP)} \quad \frac{\Gamma \vdash M : \text{Int} \quad \Gamma \vdash N : \text{Int}}{\Gamma \vdash M \text{ op } N : \text{typefor}(\text{op})} \quad \text{(VAR)} \quad \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \\
 \text{(FN)} \quad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{fn } x : A \Rightarrow M : A \rightarrow B} \quad \text{(APP)} \quad \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \\
 \text{(INL)} \quad \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl } M : A + B : A + B} \quad \text{(INR)} \quad \frac{\Gamma \vdash M : B}{\Gamma \vdash \text{inr } M : A + B : A + B} \\
 \text{(CASE)} \quad \frac{\Gamma \vdash M : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash N_1 : B \quad \Gamma, x_2 : A_2 \vdash N_2 : B}{\Gamma \vdash \text{case } M \text{ of } (\text{inl } x_1 \Rightarrow N_1 \mid \text{inr } x_2 \Rightarrow N_2) : B} \\
 \text{(PROD-INT)} \quad \frac{\Gamma \vdash M_i : A_i \quad \text{for all } 1 \leq i \leq n}{\Gamma \vdash \langle M_1, \dots, M_n \rangle : A_1 \times \dots \times A_n} \\
 \text{(PROD-ELIM)} \quad \frac{\Gamma \vdash M : A_1 \times \dots \times A_n}{\Gamma \vdash \text{proj}_i M : A_i} \quad \text{for } 1 \leq i \leq n
 \end{aligned}$$

Figure 3.1. Syntax, semantics and the typing judgements for Target.

elimination constructs for tuples and sums. The binary operators op are not specified in the syntax overview, but they can be addition (+) and subtraction (−) in the basic case, and if we add booleans to the language, comparisons (= and ≤, for example). A notable subset of the terms are *values* (V), which are significant only for the sake of defining the operational semantics or stating a Progress theorem (omitted in this thesis).

The types (A, B) in the language are similarly simple: they are either one of the base types (Int , Unit), a function type, a product type or a sum type. The inductive definition of the typing judgements $\Gamma \vdash M : A$, for Γ a variable typing context, is similar to definitions seen elsewhere. In (OP) , the type of the term depends on the binary operator in question, which is achieved using $\text{typefor}(op)$ (Int for addition).

The reduction relation $M \rightsquigarrow N$ is similarly simple, featuring reducing arithmetic expressions (using the ‘mathematical’ meaning of the operator, represented by $\llbracket op \rrbracket$), β -reduction, reductions for projections and pattern matching sums, as well as structural reduction rules – meaning reductions applied to a subexpression. The latter reduction is phrased in terms of evaluation contexts C , which are terms with *holes* in them for terms. The definition of capture-avoiding substitutions $M\{N/x\}$, as seen in the reduction rule for β -reductions, is well-known and simple.

There are some obvious possible additions to this language. The most obvious one is support for conditionals and booleans, but these can be encoded using existing constructs ($\text{Bool} = \text{Unit} + \text{Unit}$, with $\text{true} = \text{inl } \langle \rangle$ and $\text{false} = \text{inr } \langle \rangle$). We assume support for integer arithmetic in some later examples. Lists are omitted from Target (for now) for the sake of simplicity, as almost all useful list processing function require recursion.¹ When we assume the existence of lists in examples of analysis-graded monads, we also assume an interface with standard list processing functions such as extracting the head and tail, appending two lists, and so on.

3.1.2 Programming with analysis-graded monads

We now turn to the definition of analysis-graded monads and the Source language for programming with them. First we define analysis-graded monads from a programming standpoint. Then we give the syntax for the source language. Finally, we explain the translation from Source to Target, defined in Section 3.1.1.

Computational grading algebras and analysis-graded monads

As mentioned in the introduction to this chapter, an analysis-graded monad is a graded monad with an appropriate choice of grading algebra – one which represents a program analysis. We gave an overview of the abstract interpretation view of program analysis in Chapter 2. We adapt this lattice-theoretic view of program properties in order to define algebras for analysis-graded monads.

In Section 2.1 we considered several desirable components for effect algebras: an ordering relation representing *conservative overapproximation*; a multiplication operator representing the effect of running two computations *sequentially*; an *alternation* operator representing the outcome of conditional branching; a *top element* of the ordering relation representing the ‘don’t know’ outcome. We thus refer to a grading algebra corresponding to a program analysis as a *computational grading algebra*.

¹A minimal list interface might provide a list constructor and a list fold-like eliminator, as most list processing functions can be written using folding operations. While folding is itself a recursive operation, a language might provide recursion *only* for lists using folding.

Definition 3.1. A *computational grading algebra* $\mathcal{G}$ is a tuple $(E, \leq, \odot, \mathbf{1}, \mathcal{O})$, with E a set, $\leq \subseteq E \times E$ a partial order, $\odot$ a binary operator on E and $\mathbf{1} \in E$ and $\mathcal{O}$ a set such that:

- $(E, \leq)$ is a join-semilattice with join $\vee$ and a top element $\top$;
- $(E, \odot, \mathbf{1})$ is a monoid, meaning that $\mathbf{1}$ is the unit for $\odot$;
- $\odot$ is monotonic with respect to $\leq$, meaning that for all $e \leq e'$ and $f \leq f'$:

$$e \odot f \leq e' \odot f'$$

- $\mathcal{O}$ is a set of *grade operators*, which are constants drawn from E or functions with codomain E and domains $E, E \times E, \dots$ – operators on E with varying arities, *monotonic* in each argument.

More succinctly, a computational grading algebra is a pomonoid with joins and a top element, as well as function symbols of varying arities.

The join-semilattice monoid with top element structure is expected, the most unusual addition is $\mathcal{O}$. The reason for its existence is mainly for the sake of expressing graded monadic computations in the source language. This will become clearer in examples, but the main idea is that with every graded monad there are terms and functions used to reason about the effect in question. For example, with the graded monads based on `StateS`, the get operation reads from the state, so the grade present in the type of get needs to express that: in `PermStateS`, the type of get is `PermStateSro`. These notable constants in $\mathcal{O}$ (such as `ro`) are explicitly used when typing judgements are given for graded monadic operations.

A more concise presentation of $\mathcal{G}$ might state that both $\odot$ and $\mathbf{1}$ are in $\mathcal{O}$. We separate them for the sake of convenience. Furthermore, we drop $\mathcal{O}$ from the definition of computational grading algebras when discussing categorical properties.

The definition of analysis-graded monads is then brief:

Definition 3.2. An *analysis-graded monad* $\mathbf{T}$ is a monad graded with a computational grading algebra.

No further details are required, as we defined graded monads in detail in Chapter 2. All of the properties of graded monads are thus expected to hold (meaning the two unit laws, the associativity law and the monotonicity law). It remains to give a syntax for working with analysis-graded monads,

Syntax for analysis-graded monads

The common source language for programming with analysis-graded monads does not have anything that is inherently tied to analyses – instead, it is the concrete examples and instantiations that embody this link. As with `Target`, we make the language as simple as possible: it is nothing more than simply-typed lambda calculus with added constructs for programming with graded monads.

The syntax and type system of `Source` are given in Figure 3.2. The main novelty is that the language is parameterised by a particular graded monad $\mathbf{T}$ – itself defined

Syntax:

(terms) $M, N ::= n \mid \langle \rangle \mid M \text{ op } N \mid x \mid \text{fn } x : A \Rightarrow M \mid M N$
 $\quad \quad \quad \mid \text{return } M \mid \text{bind } x : A \Leftarrow M \text{ with } N \mid \text{sub}_{e,f} M$
(types) $A, B ::= \text{Int} \mid \text{Unit} \mid A \rightarrow B \mid T^e A$
(grades) $F, G ::= e \mid \mathbf{1} \mid F \odot G$
 $\quad \quad \quad \mid o(F_1, \dots, F_k) \quad (\text{for } o \in \mathcal{O} \text{ of arity } k \geq 0)$

Typing judgements:

(INT) $\frac{}{\Gamma \vdash n : \text{Int}} \quad \quad \quad (\text{UNIT}) \frac{}{\Gamma \vdash \langle \rangle : \text{Unit}}$
(OP) $\frac{\Gamma \vdash M : \text{Int} \quad \Gamma \vdash N : \text{Int}}{\Gamma \vdash M \text{ op } N : \text{typefor}(\text{op})} \quad \quad \quad (\text{VAR}) \frac{x : A \in \Gamma}{\Gamma \vdash x : A}$
(FN) $\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \text{fn } x : A \Rightarrow M : A \rightarrow B} \quad \quad \quad (\text{APP}) \frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$
(RET) $\frac{\Gamma \vdash M : A \quad \vdash_{\text{gr}} \mathbf{1} = i}{\Gamma \vdash \text{return } M : T^i A} \quad \quad \quad (\text{SUB}) \frac{\Gamma \vdash M : T^e A \quad \vdash_{\text{gr}} e \leq f}{\Gamma \vdash \text{sub}_{e,f} M : T^f A}$
(BIND) $\frac{\vdash_{\text{gr}} e \odot f = g \quad \Gamma \vdash M : T^e A \quad \Gamma, x : A \vdash N : T^f B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : T^g B}$

Figure 3.2. Syntax and the type system of Source, assuming an underlying analysis-graded monad $\mathbf{T}$ with functor (type constructor) T and grading algebra $(E, \leq, \odot, \mathbf{1}, \mathcal{O})$.

by its computational grading algebra $\mathcal{G} = (E, \leq, \odot, \mathbf{1}, O)$, a type constructor T and appropriate binding, return and subeffecting operations. Therefore the definition of the language features elements $e \in E$ used as parts of terms and types. These grades (e, f) can be considered a sort of their own in Source, but since they only range over constants, we do not consider them a separate sort in this language. However, outside of the syntax of Source, the syntax of *grade expressions* (F, G) is important: they are essentially expressions using the operators associated with the grading algebra. This means a grade expression can be a constant e from E , the unit $\mathbf{1}$ (which is just an alias for the unit element from E), sequencing or alternation of two grade expressions, or an application of a grade operator. Furthermore, we assume the existence of judgements (with possible decision procedures) for overapproximation and equality testing, capable of reasoning about the grading algebra operators (including sequencing), with a separate derivation and decision procedure:²

equality the $\vdash_{\text{gr}} F = G$ judgement merely states that, in the theory of the computational grading algebra $\mathcal{G}$, the fact that $F = G$ for $F, G \in E$.

overapproximation the $\vdash_{\text{gr}} F \leq G$ judgement merely states that, in the theory of the computational grading algebra $\mathcal{G}$, the fact that $F \leq G$ for $F, G \in E$ is derivable.

In the context of this thesis, we assume the theories of computational grading algebras are decidable and do not formalise any deductive system. Instead, the judgements can be read as simply ‘relation holds true in the monoid’.

The terms (M, N) are constants, variables, function abstractions, function applications, syntax for creating unit graded monadic computations (return), syntax for binding two graded monadic computations (`bind` $x : A \leftarrow M$ `with` N) and explicit subeffecting of graded monadic computations. The reason `sub` _{e, f} carries two grades is because we want both well-typed terms to have unique types (in a given environment) and for typing derivations of type judgements to be unique.

The types (A, B) are few: other than base types and function types, the only other type is the graded monadic $T^e A$ family. The ‘ T ’ used here refers to the *one* graded monad (more precisely, its type constructor) used for a particular instantiation of Source – support for multiple graded monads can be added by treating T as ranging over a set of graded monads and adding annotations to `return` M terms. The typing judgements $\Gamma \vdash M : A$ for the graded monadic operations rely on equality and overapproximation judgements $\vdash_{\text{gr}} F = G$ and $\vdash_{\text{gr}} F \leq G$.

The type system satisfies the following uniqueness property:

Theorem 3.1 (Uniqueness of Source typing derivations). If a Source term M is typeable under environment Γ , then it has a unique typing derivation. In other words, there exists at most one type A such that $\Gamma \vdash M : A$ and this judgement is derivable in a unique way.

Proof. We prove this statement by induction on the structure of terms M , proving the following property Φ :

$$\Phi(M) \equiv \forall \Gamma, A, B. \quad \Gamma \vdash M : A \wedge \Gamma \vdash M : B \implies A = B \text{ and the derivations of } \Gamma \vdash M : A \text{ and } \Gamma \vdash M : B \text{ match}$$

²This is similar to many similar type systems which rely on a separate judgement or theory, outside of the language. Likewise, axiomatic approaches to semantics such as Hoare Logic often assume an underlying theory of logic and arithmetic.

The proof is straightforward, so we only cover a few cases.

- (var) Let M be x . Let Γ be an environment and A, B types satisfying $\Gamma \vdash x : A$, $\Gamma \vdash x : B$. Only the (var) typing rule applies for typing judgements for terms with this syntax, so $x : A \in \Gamma$ and $x : B \in \Gamma$. Since a variable can have only one type in an environment, we conclude $A = B$. Furthermore, the assumptions match and by the hypothesis their derivations are unique, so the derivation of $\Gamma \vdash x : A$ is unique.

Thus $\Phi(x)$.

- (fn) Assume $\Phi(M)$ and the term $\text{fn } x : A \Rightarrow M$. Consider types C_1, C_2 and typing environment Γ such that $\Gamma \vdash M : C_1$ and $\Gamma \vdash M : C_2$. The only typing rule that applies to $\text{fn } x : A \Rightarrow M$ is (FN), so C_1 is of the form $A \rightarrow B_1$ and C_2 is of the form $A \rightarrow B_2$ for some types B_1 and B_2 . Furthermore, $\Gamma, x : A \vdash M : B_1$ and $\Gamma, x : A \vdash M : B_2$. By the hypothesis $\Phi(M)$, we have $B_1 = B_2$, and therefore $(A \rightarrow B_1) = (A \rightarrow B_2)$. Hence $C_1 = C_2$, and the typing derivation of $\Gamma \vdash \text{fn } x : A \Rightarrow M : C_1$ is unique by induction hypothesis.

Thus $\Phi(\text{fn } x : A \Rightarrow M)$.

- (bind) Assume $\Phi(M)$ and $\Phi(N)$ and consider the term $\text{bind } x : A \Leftarrow M \text{ with } N$. Let Γ, C_1, C_2 be such that $\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : C_1$ and $\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : C_2$. For terms of this form, the only possible last step of the derivation is (BIND), meaning that the types C_1 and C_2 are equal to $T^{g_1}B_1$ and $T^{g_2}B_2$, respectively, and necessarily there exist $A_1, A_2, e_1, e_2, f_1, f_2$ we have: $\Gamma \vdash M : T^{e_1}A$, $\Gamma, x : A_1 \vdash N : T^{f_1}B_1$, $e_1 \odot f_1 = g_1$, $\Gamma \vdash M : T^{e_2}A$, $\Gamma, x : A_2 \vdash N : T^{f_2}B_2$, $e_2 \odot f_2 = g_2$.

By assumption, we have that the types $T^{e_1}A$ and $T^{e_2}A$ are equal and that types $T^{f_1}B_1$ and $T^{f_2}B_2$ are equal.

Thus, $e_1 = e_2, f_1 = f_2, B_1 = B_2$.

It follows that $g_1 = e_1 \odot f_1 = e_2 \odot f_2 = g_2$, and hence $T^{g_1}B_1 = T^{g_2}B_2$. We conclude $C_1 = C_2$. Since the derivations of the assumptions in (BIND) are unique, the derivation of $\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : C_1$ is also unique. Hence $\Phi(\text{bind } x : A \Leftarrow M \text{ with } N)$.

□

The analogous statement also holds for Target, and the proof follow similar reasoning (and is therefore not presented here).

Theorem 3.2 (Uniqueness of Target typing derivations). If a Target term M is typeable under environment Γ , then it has a unique typing derivation. In other words, there exists at most one type A such that $\Gamma \vdash M : A$ and this judgement is derivable in a unique way.

The Source-to-Target translation

The semantics of Source is given by a translation to Target. This translation is given by three *translation functions*:

type translation this is a function $\langle\langle\cdot\rangle\rangle_{\text{type}}$ mapping Source types to Target types: for a Source type A , $\langle\langle A \rangle\rangle_{\text{type}}$ is a Target type;

environment translation this is a function $\langle\langle\cdot\rangle\rangle_{\text{env}}$ operating on Source typing environment, and it is defined entirely in terms of $\langle\langle\cdot\rangle\rangle_{\text{type}}$ – all variable types are translated from Source to Target:

$$\begin{aligned}\langle\langle\Diamond\rangle\rangle_{\text{env}} &= \Diamond \\ \langle\langle\Gamma, x : A\rangle\rangle_{\text{env}} &= \langle\langle\Gamma\rangle\rangle_{\text{env}}, x : \langle\langle A \rangle\rangle_{\text{type}}\end{aligned}$$

term translation this is a function $\langle\langle\cdot\rangle\rangle_{\text{val}}$ mapping Source terms to Target terms: for a *derivable* Source typing judgement $\Gamma \vdash M : A$, $\langle\langle\Gamma \vdash M : A\rangle\rangle_{\text{val}}$ is a term such that the following typing judgement in Target holds:

$$\langle\langle\Gamma\rangle\rangle_{\text{env}} \vdash \langle\langle\Gamma \vdash M : A\rangle\rangle_{\text{val}} : \langle\langle A \rangle\rangle_{\text{type}}$$

Even though the result of the translation $\langle\langle\Gamma \vdash M : A\rangle\rangle_{\text{val}}$ is a term and is referred to as term translation, it operates on a typing judgement. This is because the definition of this translation depends on how the structure of the term is translated and how variables in Γ are used.

The most interesting part of this translation relates to the translation of expressions with a graded monadic type. However, this aspect of the translation depends on the graded monad – it is essentially the representation of return, bind and subeffecting, which induce a graded monad. The remaining parts of the translation are not fundamentally surprising: they are agnostic of the underlying analysis-graded monad and are given in Figure 3.3. The $\langle\langle\cdot\rangle\rangle_{\text{val}}$ translation operates on *derivable* typing judgements, and fundamentally assumes the fact that there is *at most one* derivation for each typing judgement (easily established by considering the consequences of the rule instances in Figure 3.2).

One thing that remains is a result on the validity of this translation – namely, that translating a typing judgement from Source gives a derivable type judgement in Target. This result cannot be proved in general, as the translation is necessarily incomplete due to the lack of translation of terms dealing with graded monads. However, *modulo these cases*, the translation is valid – the requirements on these translations are stated in the following theorem:

Theorem 3.3. Consider the property Φ ranging over Source typing judgements $(\Phi(\Gamma \vdash M : A))$, stating the following:

$$\Phi(\Gamma \vdash M : A) \quad \equiv \quad \langle\langle\Gamma\rangle\rangle_{\text{env}} \vdash \langle\langle\Gamma \vdash M : A\rangle\rangle_{\text{val}} : \langle\langle A \rangle\rangle_{\text{type}}$$

Suppose any graded monadic operation m , which depends on terms $M_1, \dots, M_k$ and has as the unique typing rule associated with it:

$$(\tau) \frac{\Gamma \vdash M_1 : A_1 \quad \dots \quad \Gamma \vdash M_k : A_k}{\Gamma \vdash m(M_1, \dots, M_k) : t(A_1, \dots, A_k)} \quad (3.1)$$

for a type-and-grade operator t ,^a satisfies the implication:

$$\begin{aligned}\Phi(\Gamma \vdash M_1 : A_1) \wedge \dots \wedge \Phi(\Gamma \vdash M_k : A_k) \\ \implies \Phi(\Gamma \vdash m(M_1, \dots, M_k) : t(A_1, \dots, A_k)).\end{aligned}$$

$\langle\!\langle \text{Int} \rangle\!\rangle_{\text{type}}$	$=$	Int
$\langle\!\langle \text{Unit} \rangle\!\rangle_{\text{type}}$	$=$	Unit
$\langle\!\langle A \rightarrow B \rangle\!\rangle_{\text{type}}$	$=$	$\langle\!\langle A \rangle\!\rangle_{\text{type}} \rightarrow \langle\!\langle B \rangle\!\rangle_{\text{type}}$
$\langle\!\langle \Gamma \vdash n : \text{Int} \rangle\!\rangle_{\text{val}}$	$=$	n
$\langle\!\langle \Gamma \vdash \langle \rangle : \text{Unit} \rangle\!\rangle_{\text{val}}$	$=$	$\langle \rangle$
$\langle\!\langle \Gamma \vdash M \text{ op } N : \text{typefor}(\text{op}) \rangle\!\rangle_{\text{val}}$	$=$	$\langle\!\langle \Gamma \vdash M : \text{Int} \rangle\!\rangle_{\text{val}} \text{ op } \langle\!\langle \Gamma \vdash N : \text{Int} \rangle\!\rangle_{\text{val}}$
$\langle\!\langle \Gamma \vdash x : A \rangle\!\rangle_{\text{val}}$	$=$	x
$\langle\!\langle \Gamma \vdash \text{fn } x : A \Rightarrow M : A \rightarrow B \rangle\!\rangle_{\text{val}}$	$=$	$\text{fn } x : \langle\!\langle A \rangle\!\rangle_{\text{type}} \Rightarrow \langle\!\langle \Gamma, x : A \vdash M : B \rangle\!\rangle_{\text{val}}$
$\langle\!\langle \Gamma \vdash M N : B \rangle\!\rangle_{\text{val}}$	$=$	$\langle\!\langle \Gamma \vdash M : A \rightarrow B \rangle\!\rangle_{\text{val}} \langle\!\langle \Gamma \vdash N : A \rangle\!\rangle_{\text{val}}$ assuming the judgement $\Gamma \vdash M N : B$ has in its unique derivation an instance of (APP) as the last inference rules, with premises $\Gamma \vdash M : A \rightarrow B$ and $\Gamma \vdash N : A$

Figure 3.3. The type and term translation functions for the non-monadic segment of Source. The type translations follow the structure of the types, whereas the term translations follow the typing derivations. Translating application terms relies on knowing how the judgement is uniquely derived – see Theorem 3.1.

Then, $\Phi(\Gamma \vdash M : A)$ for all typing judgements $\Gamma \vdash M : A$ derivable in Source.

^aSide conditions regarding grades are not shown for the sake of brevity, but they take the form of additional assumptions in Equation 3.1.

Proof. The statement of the theorem already suggests that the proof is based on induction over derivable typing judgements, following their derivation rules. The (τ) cases, covering all additional graded monadic operations specific to a graded monad, are proved immediately by the assumption placed in the theorem. It remains to consider the other, graded monad-agnostic cases. Two simple cases are illustrated here.

(var) Let $\Gamma \vdash x : A$, meaning necessarily that $x : A \in \Gamma$.

By definition, $\langle\!\langle \Gamma \vdash x : A \rangle\!\rangle_{\text{val}} = x$. Likewise, by definition of $\langle\!\langle \cdot \rangle\!\rangle_{\text{env}}$, we have $x : \langle\!\langle A \rangle\!\rangle_{\text{type}} \in \langle\!\langle \Gamma \rangle\!\rangle_{\text{env}}$.

By (VAR) in Target, $\langle\!\langle \Gamma \rangle\!\rangle_{\text{env}} \vdash x : \langle\!\langle A \rangle\!\rangle_{\text{type}}$, or, equivalently, $\langle\!\langle \Gamma \rangle\!\rangle_{\text{env}} \vdash \langle\!\langle \Gamma \vdash x : A \rangle\!\rangle_{\text{val}} : \langle\!\langle A \rangle\!\rangle_{\text{type}}$.

(app) Assume $\Phi(\Gamma \vdash M : A \rightarrow B)$ and $\Phi(\Gamma \vdash N : A)$ for some derivable judgements $\Gamma \vdash M : A \rightarrow B, \Gamma \vdash N : A$.

By (APP) in Source, the judgement $\Gamma \vdash M N : B$ is derivable. Since this judgement has a unique derivation, $\langle\!\langle \Gamma \vdash M N : B \rangle\!\rangle_{\text{val}} = \langle\!\langle \Gamma \vdash M : A \rightarrow B \rangle\!\rangle_{\text{val}} \langle\!\langle \Gamma \vdash N : A \rangle\!\rangle_{\text{val}}$. (*)

By assumption, we have that in Target:

$$\begin{array}{lll} \langle\!\langle \Gamma \rangle\!\rangle_{\text{env}} \vdash \langle\!\langle \Gamma \vdash M : A \rightarrow B \rangle\!\rangle_{\text{val}} & : & \langle\!\langle A \rangle\!\rangle_{\text{type}} \rightarrow \langle\!\langle B \rangle\!\rangle_{\text{type}} \\ \langle\!\langle \Gamma \rangle\!\rangle_{\text{env}} \vdash \langle\!\langle \Gamma \vdash N : A \rangle\!\rangle_{\text{val}} & : & \langle\!\langle A \rangle\!\rangle_{\text{type}} \end{array}$$

Thus in Target we have, by (APP):

$$\langle\langle\Gamma\rangle\rangle_{\text{env}} \vdash \langle\langle\Gamma \vdash M : A \rightarrow B\rangle\rangle_{\text{val}} \langle\langle\Gamma \vdash N : B\rangle\rangle_{\text{val}} : \langle\langle B\rangle\rangle_{\text{type}}$$

Plugging in (*), we conclude

$$\langle\langle\Gamma\rangle\rangle_{\text{env}} \vdash \langle\langle\Gamma \vdash M N : B\rangle\rangle_{\text{val}} : \langle\langle B\rangle\rangle_{\text{type}}$$

which concludes the proof. $\square$

We complete the translation for each analysis-graded monad we come across: Target is used to describe underlying representations of graded monadic operations and types.

In short, what we are doing is we are translating typing derivation in Source to Target typing derivations. Representing this is complex, so we frame it as separately translating types, environments and terms-in-context. The term-in-context translation is phrased as operating on typing judgements, but the translation follows the (unique) derivation of this judgement.

Defining translations

In Figure 3.3 we define the type and term translation functions, and one case stands out as insufficiently defined: translating an application term requires knowing its *unique typing derivation*. In fact, the $\langle\langle\cdot\rangle\rangle_{\text{val}}$ definition follows typing derivations of judgements, and consequently we adopt notation that makes this explicit and avoids awkward side conditions like in the application case.

Suppose that, in a type derivation, a typing judgement $\Gamma \vdash M : A$ is derived from typing judgements $\Gamma_1 \vdash N_1 : B_1, \dots, \Gamma_k \vdash N_k : B_k$:

$$\frac{\Gamma_1 \vdash N_1 : B_1 \quad \dots \quad \Gamma_k \vdash N_k : B_k}{\Gamma \vdash M : A}$$

Then the translation $\langle\langle\Gamma \vdash M : A\rangle\rangle_{\text{val}}$ depends on $t_1 = \langle\langle\Gamma_1 \vdash N_1 : B_1\rangle\rangle_{\text{val}}, \dots, t_k = \langle\langle\Gamma_k \vdash N_k : B_k\rangle\rangle_{\text{val}}$ using a *combinator* $\mathcal{K}[t_1, \dots, t_k]$. We adopt as notation the abbreviation $t_1, \dots, t_k$ to stand for $\langle\langle\Gamma_1 \vdash N_1 : B_1\rangle\rangle_{\text{val}}, \dots, \langle\langle\Gamma_k \vdash N_k : B_k\rangle\rangle_{\text{val}}$ whenever we are dealing with a rule with such premises. The notation we use to define this translation and combinator is as follows:

$$\frac{\Gamma_1 \vdash N_1 : B_1 \quad \dots \quad \Gamma_k \vdash N_k : B_k}{\Gamma \vdash M : A} \bullet \mathcal{K}[t_1, \dots, t_k]$$

For example, the application translation before can be more precisely be written as:

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} \bullet t_1 t_2$$

In this notation we ignore all side conditions when listing $t_1, \dots, t_k$, since they are not valid inputs for $\langle\langle\cdot\rangle\rangle_{\text{val}}$.

In further examples, we mostly use this notation, though occasionally (when the inference rule is clear and unambiguous) we use the convention adopted in Figure 3.3.

3.1.3 Haskell embedding

So far in this section, analysis-graded monads have been phrased in the context of a graded monadic source language *Source* and a target language *Target*. However, they can be implemented in Haskell using existing features – with some degree of difficulty, as it turns out.

Orchard and Petriček’s effect-monad [OP14] Haskell package is cited in literature (for example, by Gaboardi et al. [GKO⁺16] and by Bracker and Nilsson [BN15]) as a way of embedding graded monads corresponding to effects into Haskell. It is a simple and elegant representation of various theoretical graded monadic frameworks. Unfortunately, the core definition has limitations that prevent full representation of graded monads (for example, a precise analysis of whether a computation fails or succeeds) – as we will see in examples of analysis-graded monads. To address these shortcomings, I introduce the *analysis-monad* package intended for representing analysis-graded monads. This Section summarises this implementation. The details of the implementation, as well as the source code, are given in the Appendix – though the package is available on GitHub as well:

<https://github.com/Dreian/analysis-monad>

The *analysis-monad* package is inspired by *effect-monad*, so some of the definitions and signatures are almost identical. I do not assume familiarity with prior embeddings of effect systems in Haskell, but I stress the novel aspects of *analysis-monad* and their significance.

To start, every analysis-graded monad should be an instance of a type class *GMonad* defined in this library, which requires a definition of the underlying grading algebra (encoded as a separate kind, and kinds are encoded as types using *DataKinds*), as well as of the main graded monadic operations (return, bind, effect upcasting/subeffecting). Notably, we impose the condition that the grading algebra has a *join* operator, used in the grade of a conditional branching expression. The definition of the type class starts in the following way:

```
class GFunctor m => GMonad (m :: k -> * -> *) where
  -- grading algebra
  type Unit m :: k
  type Seq m (e :: k) (f :: k) :: k
  type Join m (e :: k) (f :: k) :: k
  ...
```

The one detail in this code is the requirement that *m* is also a *family of functors*, which is achieved using a separate *GFunctor* type class. This is a simple variant of Haskell’s *Functor* type class, with *fmapG* being the graded version of *fmap*:

```
class GFunctor (f :: k -> * -> *) where
  fmapG :: (a -> b) -> f r a -> f r b
```

Notably, the partial order of the grading algebra is not directly present in the definition of *GMonad*. Instead, subeffecting is achieved using a multi-parameter *Subeffect* type class, where *Subeffect m e f* for a combination of *m*, *e*, *f* signifies that *e* is a subeffect of *f* and both are elements of the grading algebra associated with *m*. This type class contains the explicit coercion operator:

```
class Subeffect (m :: k -> * -> *) (e :: k) (f :: k) where
  sub :: m e a -> m f a
```

The type system of Haskell allows the programmer to use `sub` in many contexts, though it typically requires typing annotations in order to specify which grade the argument is upcasted to. For example, the aforementioned branching is implemented using the following simple definition:

```
alt :: JoinUpper m e f
    => Bool -> m e a -> m f a -> m (Join m e f) a
alt True  x y = sub x
alt False x y = sub y
```

Here a constraint is used in order to represent the fact the join of `e` and `f` is an upper bound of both in the grading algebra for graded monad `m`. This is just the following composite constraint:

```
type JoinUpper m e f = (Subeffect m e (Join m e f),
                        Subeffect m f (Join m e f))
```

Finally, the `GMonad` type class defined in the library requires definitions of `return` and `binding`. While `return` has a simple signature, `(>=)` does not. In a graded monad, the graded bind is a family of operations for each pair of grades. This means that a sophisticated definition of an analysis-graded monad might need to consider the grades associated with both of the computations being sequenced. However, it is impossible to pattern match on a term of a function type. Thus, a proper implementation of binding requires the use of *ad hoc* polymorphism, where while computing `m >= k` the appropriate version of binding is chosen depending on the types of both `m` and `k`. In Haskell, such polymorphism is achieved using type classes, and in this case we use a type class called `GBindable`. This type class requires a `bindG` function which enables ad-hoc polymorphism required when defining binding.

```
...
return :: a -> m (Unit m) a
(>=) :: GBindable m e f
      => m e a -> (a -> m f b) -> m (Seq m e f) b
x >= k = bindG x k

class GMonad m
  => GBindable (m :: k -> * -> *) (r :: k) (s :: k) where
  bindG :: m r a -> (a -> m s b) -> m (Seq m r s) b
```

This implementation of binding is the main way in which analysis-monad differs from effect-monad.

Showing that a particular type constructor is an analysis-graded monad, it suffices to do the following:

Encode the grading algebra via types Devise a representation of the grading algebra carrier set using a type-level encoding, and its join and sequencing operators using type families or similar constructs.

Implementing all subeffecting This means defining, for all $e \leq f$, an instance of `Subeffect m e f` and the associated sub functions.

Show it is a family of functors In other words, show that the graded monad is an instance of the `GFunctor` type class by implementing `fmapG` for each grade.

Implement the return operator This means showing that it is an instance of `GMonad`, which includes implementing `return` and stating the main components of the grading algebra.

Implement the family of binding operators This means, for all pairs of grades, defining the corresponding `bindG` function:

```
bindG :: GMonad m => m r a -> (a -> m s b) -> m (Seq m r s) b
```

In addition, there is occasionally need to rely on properties of a particular graded monad, or all graded monad in particular. Such properties are expressed using Haskell constraints. For example, associativity of the sequencing operator can be expressed using the following equality constraint:

```
type Assoc m e f g = ((Seq m (Seq m e f) g)
                      ~ (Seq m e (Seq m f g)))
```

3.2 Computing with failure

One of the simplest monads introduced to a student of functional programming is **Maybe**: an expression whose type is **Maybe A** represents a computation that might fail, but if it does not and it terminates, then it results in a value whose type is *A*. This monad is used in functional languages in place of exceptions or explicit error codes. Its graded version **GMaybe** was the simplest graded monad considered in Chapter 2 – its grading algebra has a carrier set $\{T, F, U\}$, and these items represent respectively the information that the computation is guaranteed to succeed, that it is guaranteed to fail, and that there are no (statically decidable) guarantees. An extension of **Maybe** called **Except** is sometimes used to more specifically represent exceptions and allow syntax similar to that of ML-like languages ([handle](#), [raise](#)). The corresponding graded monad **GExcept** has a grade that represents the set of all exceptions that the graded monad might raise. This way, the analysis performed by **GExcept** resembles that of *checked exceptions* seen in Java, where all functions that might raise (and not handle) a checked exception must state the exceptions they might raise.

These simple analyses are instructive because properly implementing them in existing programming languages is not straightforward. In fact, the most notable implementation of graded monads in Haskell, Orchard’s effect-monad, oversimplifies the grading algebra associated with **GMaybe** in order to keep the library simple (there are only).

In this Section, I give the translations of these two monads into Target and also provide their Haskell embedding. The implementation of **GMaybe** in Haskell in particular informs the design decisions of `analysis-monad`.

3.2.1 Deciding safety and failure

The grading algebra associated with **GMaybe**, $(\{T, F, U\}, \sqsubseteq, \odot, T, \{F\})$, defines its sequencing operator $\odot$ over $\{T, F, U\}$ as follows:

$$\begin{aligned} T \odot e &= e \\ F \odot e &= F \\ U \odot F &= F \\ U \odot e &= U \quad (\text{assuming } e \neq F) \end{aligned}$$

The most interesting case is the third one, $U \odot F = F$. This means that, when sequencing two computations, of which the latter fails, the sequenced computation

fails. This differs from the implementation in `effect-monad`, where $\mathbf{U} \odot \mathbf{F}$ is defined to be $\mathbf{U}$.

I first give a formal treatment of this analysis-graded monad and give its translation into `Target`, and then show how to implement it in Haskell.

Operation and translation

Suppose the graded monad T in `Source` is `GMaybe`. In addition to representing successful computations, a term of this type might represent a failing computation. We add a term `nothing` : A to the syntax of `Source` to represent such failure (along with a type annotation to ensure uniqueness of typing), along with the following typing rule:

$$(\text{NOTHING}) \frac{}{\Gamma \vdash \text{nothing} : A : \text{GMaybe}^{\mathbf{F}} A}$$

The type system of the language is equipped to reason about safe and unsafe computations. For example, the term:

`bind` $x : \text{Int} \Leftarrow \text{return } 5 \text{ with } \text{return } x + 1$

is deemed safe, as its type is `GMaybeTInt` (it is a computation terminating with the number 6), whereas the type of the following unsafe computation is `GMaybeFInt`:

`bind` $x : \text{Int} \Leftarrow (\text{nothing} : \text{Int}) \text{ with } \text{return } x + 1$

An addition of conditional branching to `Source` can provide examples of statically undecidable problems:

$b : \text{Bool} \vdash \text{if } b \text{ then } \text{return } 0 \text{ else } (\text{nothing} : \text{Int}) : \text{GMaybe}^{\mathbf{U}} \text{Int}$

It is necessary to define the translation functions $\langle\langle \cdot \rangle\rangle_{\text{type}}$ and $\langle\langle \cdot \rangle\rangle_{\text{val}}$ for `GMaybe` in order to describe its semantics. This means defining $\langle\langle \text{GMaybe}^e A \rangle\rangle_{\text{type}}$ for every $e \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}$ and the translations of `return`, `nothing` : A , and binding and subeffecting for every pair of grades. This translation is given in Figure 3.4. Translating the types follows the `GMaybe` definition seen in Chapter 2 from a categorical point of view. The subtyping and binding translations are not difficult, with the most involved case being binding corresponding to $\mathbf{U} \odot e = \mathbf{U}$ (when $e \neq \mathbf{F}$). In this case, the translation to `Target` needs to look at the outcome of the first computation, and depending on this return either $\langle \rangle$ (corresponding to `nothing`) or sequence the two computations.

Note that the translation of `GMaybeUA` matches exactly the way `Maybe` is represented, and $\mathbf{U}$ is that top element representing the ‘don’t know’ analysis outcome. In a sense, all grades below $\mathbf{U}$ (in general $\top$) represent a useful special case elaborating (or restricting) the semantics of a non-graded monadic computation. We see this in other examples of analysis-graded monads, and there is a useful theoretical foundation of this question.

A sensible addition to this version of `GMaybe` would be a `runGMaybe` operation which, for a given possibly failing computation and a default result, returns either the successful answer or the default answer in case the computation fails. The translation of this construct dispatches on the various grades the computation might have.

$$\begin{aligned}
\langle\langle \text{GMaybe}^T A \rangle\rangle_{\text{type}} &= \langle\langle A \rangle\rangle_{\text{type}} \\
\langle\langle \text{GMaybe}^F A \rangle\rangle_{\text{type}} &= \text{Unit} \\
\langle\langle \text{GMaybe}^U A \rangle\rangle_{\text{type}} &= \langle\langle A \rangle\rangle_{\text{type}} + \text{Unit}
\end{aligned}$$

$$\begin{aligned}
(\text{RET}) \quad & \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{return } M : \text{GMaybe}^T A} & \bullet & t_1 \\
(\text{NOTHING}) \quad & \frac{}{\Gamma \vdash \text{nothing} : A : \text{GMaybe}^F A} & \bullet & \langle \rangle \\
(\text{BIND-T}) \quad & \frac{\Gamma \vdash M : \text{GMaybe}^T A \quad \Gamma, x : A \vdash N : \text{GMaybe}^e B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{GMaybe}^e B} & \bullet & t_2\{t_1/x\} \\
(\text{BIND-F}) \quad & \frac{\Gamma \vdash M : \text{GMaybe}^F A \quad \Gamma, x : A \vdash N : \text{GMaybe}^e B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{GMaybe}^e B} & \bullet & \langle \rangle \\
(\text{BIND-U1}) \quad & \frac{\Gamma \vdash M : \text{GMaybe}^U A \quad \Gamma, x : A \vdash N : \text{GMaybe}^F B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{GMaybe}^F B} & \bullet & \langle \rangle \\
(\text{BIND-U2}) \quad & \frac{\Gamma \vdash M : \text{GMaybe}^U A \quad \Gamma, x : A \vdash N : \text{GMaybe}^G B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{GMaybe}^F B} & \bullet & \\
& \text{case } t_1 \text{ of } (\text{inl } x_1 \Rightarrow t_2\{x_1/x\} \mid \text{inr } x_2 \Rightarrow \langle \rangle) & (G \neq F) \\
(\text{SUB-REFL}) \quad & \frac{\Gamma \vdash M : \text{GMaybe}^e A}{\Gamma \vdash \text{sub}_{e,e} M : \text{GMaybe}^e A} & \bullet & t_1 \\
(\text{SUB-T-U}) \quad & \frac{\Gamma \vdash M : \text{GMaybe}^T A}{\Gamma \vdash \text{sub}_{T,U} M : \text{GMaybe}^U A} & \bullet & \text{inl } t_1 : \langle\langle A \rangle\rangle_{\text{type}} + \text{Unit} \\
(\text{SUB-F-U}) \quad & \frac{\Gamma \vdash M : \text{GMaybe}^F A}{\Gamma \vdash \text{sub}_{F,U} M : \text{GMaybe}^U A} & \bullet & \text{inr } t_1 : \langle\langle A \rangle\rangle_{\text{type}} + \text{Unit}
\end{aligned}$$

Figure 3.4. The translation of **GMaybe** into Target. Whenever a new bound variable is introduced, we assume a fresh name is used.

Haskell embedding

The fact that the resulting grades for $U \odot F$ and $U \odot G$ (for $G \neq F$) differ is one of the simplest examples why analysis-monad introduces `GBindable` for implementing its binding operation.

Looking at the effect-monad implementation, the grading definition of the grading algebra associated with the similar `IMaybe` graded monad is as follows (where `Plus` plays the same role as our `Seq`):

```
...
type Plus IMaybe F s = F
type Plus IMaybe T s = s
type Plus IMaybe U s = U
...
```

The lack of distinct case $U \odot F = F$ means that the analysis given by `IMaybe` is less precise than what we seek.

The `IMaybe` graded monad is implemented as a generalised algebraic data type (GADT), meaning there is a separate data constructor for each grade:

```
data IMaybe p a where
  INothing ::          IMaybe F a
  IJust    :: a        -> IMaybe T a
  IDyn     :: IMaybe s a -> IMaybe U a
```

The bind operator $\gg=$ is a function that operates on two terms, M and N , whose types are $T^e A$ and $A \rightarrow T^f B$, respectively. While it is possible to perform pattern matching on M , pattern matching on data of a function type is impossible. Each part of the definition of $(\gg=)$ must have the appropriate type of the result, and a `case` statement is of no use. Hence, the implementation of $(\gg=)$ when the first argument has grade U in effect-monad is as follows:

```
(IDyn (IJust a)) >>= k = IDyn (k a)
(IDyn (INothing)) >>= k = IDyn INothing
```

In the case of analysis-monad, we define a similar graded monad `GMaybe`:

```
data GMaybe r a where
  GNothing ::          GMaybe F a
  GJust    :: a        -> GMaybe T a
  GDyn     :: GMaybe r a -> GMaybe U a
```

Then the binding operator is defined by showing how to perform binding for `GMaybe` for each pair of grades in the following ways.

```
instance GBindable GMaybe F s where
  bindG GNothing _ = GNothing
instance GBindable GMaybe T s where
  bindG (GJust x) k = k x
instance GBindable GMaybe U F where
  bindG (GDyn x) _ = GNothing
instance GBindable GMaybe U T where
  bindG (GDyn GNothing) _ = GDyn GNothing
  bindG (GDyn (GJust x)) k = GDyn (k x)
instance GBindable GMaybe U U where
  bindG (GDyn GNothing) _ = GDyn GNothing
  bindG (GDyn (GJust x)) k = GDyn (k x)
```

One thing we can see is that the definition of `bindG` closely resembles that of $\langle\langle \Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{GMaybe}^e B \rangle\rangle_{\text{val}}$ in Figure 3.4.

3.2.2 Checked exceptions

Representation and translation

The grading algebra of the graded monad **GExcept** must take into account all of the possible finite number of exceptions that exist in a program – call it *Exceptions*. A term whose type is **GExcept**^{*e*} *A*, with $e \subseteq \text{Exceptions}$, is the type of computations for which the set of unhandled exceptions is *e*. The carrier set of the algebra is thus $\mathcal{P}(\text{Exceptions})$ and the ordering in the algebra is $\subseteq$. Sequencing two computation merely takes the union of the two sets of exceptions, and this operation has $\emptyset$ as its identity. Raising a particular exception ϵ has a graded monadic type whose grade is a singleton set $\{\epsilon\}$, whereas handling an exception removes it from the set of exceptions. The grading algebra is thus:

$$(\mathcal{P}(\text{Exceptions}), \subseteq, \cup, \emptyset, \bigcup_{\epsilon \in \text{Exceptions}} \{\{\epsilon\}, \text{hdl}_\epsilon\})$$

where hdl_ϵ is a function that takes the grade of a computation and the grade of its handler and computes the grade of the entire computation: $\text{hdl}_\epsilon(e, f) = (e \setminus \{\epsilon\}) \cup f$.

Extending Source with support for exception handling is simple – we add syntax for raising and for handling exceptions: $\text{raise}_\epsilon : A$ (with a type annotation necessary for uniqueness of typing) and $\text{handle}_\epsilon M N$, where *M* is the tried computation, and *N* is the computation run in case *M* raises and does not handle ϵ . Their typing judgements are simple:

$$\begin{array}{c} \text{(RAISE)} \quad \frac{}{\Gamma \vdash \text{raise}_\epsilon : A : \text{GExcept}^{\{\epsilon\}} A} \\ \\ \text{(HANDLE)} \quad \frac{\Gamma \vdash M : \text{GExcept}^e A \quad \Gamma \vdash N : \text{GExcept}^f A \quad \vdash \text{hdl}_\epsilon(e, f) = g}{\Gamma \vdash \text{handle}_\epsilon M N : \text{GExcept}^g A} \end{array}$$

The reason why handle_ϵ is not seen as a function is because neither Source nor Target feature any polymorphism over grades. Grade polymorphism is a more involved question, which we turn to later in this thesis.

When it comes to the translation to Target, the most difficult part is the underlying data representation. There are multiple ways of doing this and the Haskell embedding later shows a different (arguably more elegant) solution. The fact that *Exceptions* is finite provides a way to encode them: we assign an ordering to all of the exceptions $\epsilon_1, \dots, \epsilon_m$ and in Target we use the corresponding exception number to refer to it. This means that for every grade the translation of **GExcept**^{*e*} *A* is nothing more than a sum type $\langle\langle A \rangle\rangle_{\text{type}} + \text{Int}$. The remainder of the translation is similar to **GMaybe** and is given in Figure 3.5. In this translation, we assume Target has syntactic sugar and the ability to test integer equality, which is needed for proper exception handling.

A reasonable addition to Source would be a way of running the computation, but only in those cases when it is guaranteed to not raise and leave unhandled any exceptions (meaning that the grade is $\emptyset$).

Haskell embedding

This graded monad can be embedded into Haskell because grades, which are sets, can be neatly represented at the type level, as shown by Orchard in the

$$\begin{aligned}
& \langle\langle \text{GExcept}^e A \rangle\rangle_{\text{type}} = \langle\langle A \rangle\rangle_{\text{type}} + \text{Int} \\
& (\text{RET}) \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{return } M : \text{GExcept}^0 A} \quad \sim \quad \text{inl } t_1 : \langle\langle A \rangle\rangle_{\text{type}} + \text{Int} \\
& (\text{RAISE}) \frac{}{\Gamma \vdash \text{raise}_{\epsilon_i} : A : \text{GExcept}^{\{\epsilon_i\}} A} \quad \sim \quad \text{inr } i : \langle\langle A \rangle\rangle_{\text{type}} + \text{Int} \\
& (\text{SUB}) \frac{\Gamma \vdash M : \text{GExcept}^e A}{\Gamma \vdash \text{sub}_{e,f} M : \text{GExcept}^f A} \quad \sim \quad t_1 \\
& (\text{HANDLE}) \frac{\Gamma \vdash M : \text{GExcept}^e A \quad \Gamma \vdash N : \text{GExcept}^f A}{\Gamma \vdash \text{handle}_{\epsilon_i} M N : \text{GExcept}^{\text{hdl}_{\epsilon_i}(e,f)} A} \quad \sim \quad \bullet \\
& \text{case } t_1 \text{ of } (\text{inl } x \Rightarrow t_1 \mid \text{inr } y \Rightarrow \text{if } y = i \text{ then } t_2 \text{ else } t_1) \\
& (\text{BIND}) \frac{\Gamma \vdash M : \text{GExcept}^e A \quad \Gamma, x : A \vdash N : \text{GExcept}^f B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{GExcept}^{e \cup f} B} \quad \sim \quad \bullet \\
& \text{case } t_1 \text{ of } (\text{inl } x_1 \Rightarrow t_2\{x_1/x\} \mid \text{inr } x_2 \Rightarrow t_1)
\end{aligned}$$

Figure 3.5. The translation of **GExcept** into Target. As before, we assume a fresh name for every new bound variable.

type-level-sets library.³ Type-level sets are represented as type-level lists without duplicates. Defining the effect algebra for checked exceptions is done by having the programmer define a kind whose inhabitants are all exceptions, but also an exception type used in the underlying representation of `GExcept`. Assuming there are three interesting checked exceptions in the program (index out of bounds, division by zero, and an exception with an associated string message), such a definition would look like this:

```
data Exceptions = OutOfBounds | DivByZero | CustomExc
data ExceptionsE = OutOfBoundsE | DivByZeroE | CustomExcE String
type GExcept (r :: [Exceptions]) a = Either ExceptionsE a
```

Defining `bind`, `return` and other components that demonstrate that `GExcept` is a graded monad is not difficult. What is particularly interesting is defining the `raise` and `handle` computations which are used for working with exceptions. In the absence of dependent types, each exception requires a separate `raise` and `handle`, for example `raise00B` and `handleDBZ` (respectively standing for ‘out of bounds’ and ‘divide by zero’). In a way, what one would expect to be a function argument is encoded into the function name. The following is an implementation of all of the `raise` functions:

```
raise00B :: GExcept '[OutOfBounds] a
raise00B  = Left OutOfBoundsE
raiseDBZ :: GExcept '[DivByZero] a
raiseDBZ  = Left DivByZeroE
raiseCE   :: GExcept '[CustomExc] a
raiseCE e = Left (CustomExcE e)
```

All of these definitions require manually stating which exception is the one being raised. Handling also requires such annotations and is more complicated: handling an exception removes it from the set of unhandled exceptions, but allowing for the possibility of it being re-raised by the handling behaviour.

```
handle00B :: GExcept r a -> GExcept s a
            -> GExcept (Union (r :\ OutOfBounds) s) a
handle00B (Left OutOfBoundsE) h = h
handle00B m _ = m
handleDBZ :: GExcept r a -> GExcept s a
            -> GExcept (Union (r :\ DivByZero) s) a
handleDBZ (Left DivByZeroE) h = h
handleDBZ m _ = m
handleCE   :: GExcept r a -> (e -> GExcept s a)
            -> GExcept (Union (r :\ CustomExc) s) a
handleCE (Left (CustomExcE e)) h = h e
handleCE m _ = m
```

The grading algebra can be slightly improved to express the fact that an expression is definitely raised and not handled: the carrier set can be that of functions from exception names to a semilattice similar to the one seen for `GMaybe`. We omit this extension because it is not instructive or a major improvement, and it is not an analysis seen in existing ways of reasoning about checked exceptions.

Related work

There are several existing type systems and effects systems used for detecting sets of uncaught exceptions. The earliest one seems to be a type-based analysis for an

³<https://hackage.haskell.org/package/type-level-sets>

ML-like language by Leroy and Pessaux [LP00]. Benton and Buchlovsky [BB07b] give a monadic semantics of an effect analysis for exceptions where monadic expressions are wrapped inside monads – without using the language of graded monads, even though the fundamental idea is similar ($T_\varepsilon A$ is the type of exception-raising computations with result of type A , and ε represents the set of possibly unhandled exceptions).

3.3 Stateful computation

Analysing stateful computation is one of the most common settings for program rewriting. Therefore it is not surprising that these have corresponding analysis-graded monads.

I first consider the analysis-graded monad based on the permissions example (Example 2.3). Next, I show how to implement and represent several analysis-graded monads whose grades carry information about how many times a state is read from or modified – which find use when reasoning about resource usage. Finally, I show how one of the classical program analyses, *live variable analysis*, can be represented using graded monads using an appropriate grading algebra.

3.3.1 Permissions

In Chapter 2, one of the examples of graded monads was `PermState`, in which the grade represents the capabilities a computation has over a particular state. The grading algebra is a five-element join-semilattice, and the grades represent the possible semantics of a stateful computation: pure (`pure`), read-only (`ro`), always-write (`aw`), write-only (`wo`) and unrestricted/read-write (`rw`). In a practical setting, this graded monad can be used to provide a programmer with guarantees, for example that a function (or any kind of computation) does not mutate a state. Note that the top element of the lattice, `rw`, represents the ‘don’t know’ outcome, which matches the intuition introduced in Section 3.1.

Since this example was given attention in Chapter 2, the exact definition of the analysis-graded monad in question is not repeated here. Instead, we turn immediately to the question of implementing this analysis-graded monad in a programming language: both through the Source to Target translation and with the Haskell embedding.

Representation and translation

Programming with a stateful computation assumes the existence of a `get` computation and a `put` function. In addition, we introduce `run M` syntax to represent ‘running’ or ‘unwrapping’ a graded monadic stateful computation, resulting in a function that computes the final result of the computation based on the initial value of the state. In Source, the graded monad for stateful computations is `PermStateS`

for some type S , and the typing judgements for these computations are as follows:

$$\begin{aligned}
 (\text{GET}) \quad & \frac{}{\Gamma \vdash \text{get} : \text{PermState}_S^{\text{ro}} S} \\
 (\text{PUT}) \quad & \frac{}{\Gamma \vdash \text{put} : S \rightarrow \text{PermState}_S^{\text{aw}} \text{Unit}} \\
 (\text{RUN}) \quad & \frac{\Gamma \vdash M : \text{PermState}_S^e A}{\Gamma \vdash \text{run } M : S \rightarrow A}
 \end{aligned}$$

The grading algebra is thus $(\{\text{pure}, \text{ro}, \text{aw}, \text{wo}, \text{rw}\}, \sqsubseteq, \odot, \text{pure}, \{\text{ro}, \text{aw}\})$. Programming with such a graded monad is then done in exactly the same way as with the usual State_S monads in any pure functional language.

The translation of PermState_S into Target follows the definition of the graded monad in category theory (Section 2.3). The translation of the five graded monadic types, for each grade, is expressed using function, product and sum types. Encoding `get` and `put` is simple and based on the definitions seen for State_S , except in a graded setting. The translation of `run` is somewhat more involved, as it needs to consider all of the possible grades for the underlying stateful computation. The main challenge comes from the binding, whose translation is complicated. It is not particularly difficult, it just requires distinguishing a large number of cases. The most interesting parts of the translation are given in Figure 3.6, whereas the remainder of the translation is given in the Appendix.

The difficulty of translating `bind` $x : A \leftarrow M$ `with` N and `sube,f` M can be significant, as we see in this instance. All of the other analysis-graded monads we consider have much simpler translations. In the worst case, a separate translation rule must be given for all pairs of grades $\langle e, f \rangle$. Thus, if the grading algebra has k elements, this leads to $O(k^2)$ cases. Thankfully, in most practical cases there are useful shortcuts – even for infinite grading algebras.

Haskell embedding

This example is another one that had to be simplified in prior work: effect-monad only considers grades `pure`, `ro` and `rw` because otherwise implementing graded `bind` is difficult. The analysis-monad library aims to follow the definition of this analysis-graded monad completely.

The `PermState s` analysis-graded monad has the underlying representation:

```

data PermState s (f :: Perm) a where
  PureState :: {runPureState :: a}
              -> PermState s Pure a
  ROState   :: {runROState    :: s -> a}
              -> PermState s RO   a
  AWState   :: {runAWState    :: (a, s)}
              -> PermState s AW   a
  WOState   :: {runWOState    :: (a, Maybe s)}
              -> PermState s WO   a
  RWState   :: {runRWState    :: s -> (a, s)}
              -> PermState s RW   a

```

This means that the main stateful computations get defined in the following way:

```

get :: PermState s RO s
get = ROState id

```

$$\begin{aligned}
\langle\langle \text{PermState}_S^{\text{pure}} A \rangle\rangle_{\text{type}} &= \langle\langle A \rangle\rangle_{\text{type}} \\
\langle\langle \text{PermState}_S^{\text{ro}} A \rangle\rangle_{\text{type}} &= \langle\langle S \rangle\rangle_{\text{type}} \rightarrow \langle\langle A \rangle\rangle_{\text{type}} \\
\langle\langle \text{PermState}_S^{\text{aw}} A \rangle\rangle_{\text{type}} &= \langle\langle A \rangle\rangle_{\text{type}} \times \langle\langle S \rangle\rangle_{\text{type}} \\
\langle\langle \text{PermState}_S^{\text{wo}} A \rangle\rangle_{\text{type}} &= \langle\langle A \rangle\rangle_{\text{type}} \times (\langle\langle S \rangle\rangle_{\text{type}} + \text{Unit}) \\
\langle\langle \text{PermState}_S^{\text{rw}} A \rangle\rangle_{\text{type}} &= \langle\langle S \rangle\rangle_{\text{type}} \rightarrow (\langle\langle A \rangle\rangle_{\text{type}} \times \langle\langle S \rangle\rangle_{\text{type}})
\end{aligned}$$

$$\begin{aligned}
(\text{RET}) \quad & \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{return } M : \text{PermState}_S^{\text{pure}} A} \quad \bullet \quad t_1 \\
(\text{GET}) \quad & \frac{}{\Gamma \vdash \text{get} : \text{PermState}_S^{\text{ro}} S} \quad \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow s \\
(\text{PUT}) \quad & \frac{}{\Gamma \vdash \text{put} : S \rightarrow \text{PermState}_S^{\text{aw}} \text{Unit}} \quad \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow \langle\langle \rangle, s \rangle \\
(\text{RUN-pure}) \quad & \frac{\Gamma \vdash M : \text{PermState}_S^{\text{pure}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} \quad \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow t_1 \\
(\text{RUN-ro}) \quad & \frac{\Gamma \vdash M : \text{PermState}_S^{\text{ro}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} \quad \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow t_1 s \\
(\text{RUN-aw}) \quad & \frac{\Gamma \vdash M : \text{PermState}_S^{\text{aw}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} \quad \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow \text{proj}_1 t_1 \\
(\text{RUN-wo}) \quad & \frac{\Gamma \vdash M : \text{PermState}_S^{\text{wo}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} \quad \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow \text{proj}_1 t_1 \\
(\text{RUN-rw}) \quad & \frac{\Gamma \vdash M : \text{PermState}_S^{\text{rw}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} \quad \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow \text{proj}_1 (t_1 s) \\
(\text{BIND-ro-aw}) \quad & \frac{\Gamma \vdash M : \text{PermState}_S^{\text{ro}} A \quad \Gamma, x : A \vdash N : \text{PermState}_S^{\text{aw}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{GMaybe}^{e \cup f} B} \\
& \bullet \quad \text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow t_2 \{(t_1 s)/x\}
\end{aligned}$$

Figure 3.6. The translation of PermState_S into Target. Fresh names are assumed for all introduced variables in translations (such as s in function abstractions seen in translations to Target). Only one case for binding is shown (the one corresponding to $\text{ro} \odot \text{aw} = \text{rw}$) and no cases for subeffecting is given. See Appendix for the full translation.

```
put :: s -> PermState s AW ()
put s = AWState ((), s)
```

The sequencing operator of this graded monad is represented directly as a Haskell type family. The major challenge of implementing `PermState s` lies in implementing `bind` – there are twenty-one instances of the `GBindable` type class used for fully defining it, which includes cases such as the following:

```
instance GBindable (PermState s) R0 AW where
  bindG (R0State f) k = RWState (runAWState . k . f)
```

This are the main components of the definition of `PermState`, and under certain conditions the grades of computations can get inferred automatically. For example, the type of function `foo`:

```
foo x b = if b then put (x + 1) else return ()
```

is correctly inferred to be `Int -> Bool -> PermState Int W0 ()`.

3.3.2 Live variable analysis

One of the most important classical program analyses, which motivated the development of more general approaches such as *monotone data flow analysis* and abstract interpretation, is *live variable analysis*. Traditionally, live variable analysis is performed on imperative programs, seen as control-flow graphs. We say that a mutable variable is *live* at a program point (entry or exit to a node) if there is a path in the control graph in which the value of the mutable variable is read from prior to being written to, and otherwise we say that it is *dead*. The analysis associates with each program point a set of names of mutable variables which are live at that point (see left part of Figure 3.7). When phrased this way, live variable analysis is a safe overapproximation of the undecidable program property of reading a variable prior to writing it for all possible execution traces. Live variable analysis is commonly used for identifying dead code, which are instructions which can be removed from the program while preserving program semantics.

In a functional setting, computations utilising mutable state are represented using (graded) monads. While the question of optimising monadic computations is an interesting one, for us live variable analysis is interesting because it provides a guarantee and a *contract* to the programmer (just like types). Live variable information is associated with a computation as opposed to a program point or an individual instruction, so saying that variables `Y` and `Z` are live in a computation `c` and `x` is dead means that the programmer should care about what the values of `Y` and `Z` are immediately before `c` is run, but the value of `x` immediately prior to the computation is irrelevant.

The first step is to devise a grading algebra that represents live variable analysis.

Algebra of transfer functions

Live variable analysis in an imperative sense is performed by generating a set of subset constraints and finding the least, most precise solution. For example, each assignment generates a constraint that captures the dependence of live-in variables *LiveIn* on the live-out variables *LiveOut*. Assuming instruction `n` is the assignment `X := Y + Z`, the constraint associated with it is $LiveIn(n) = (LiveOut(n) \setminus \{X\}) \cup \{Y, Z\}$. Similar relationships exist whatever the instruction is, the live-in set is always a function of the live-out set (hence we say that live variable analysis is a *backward*

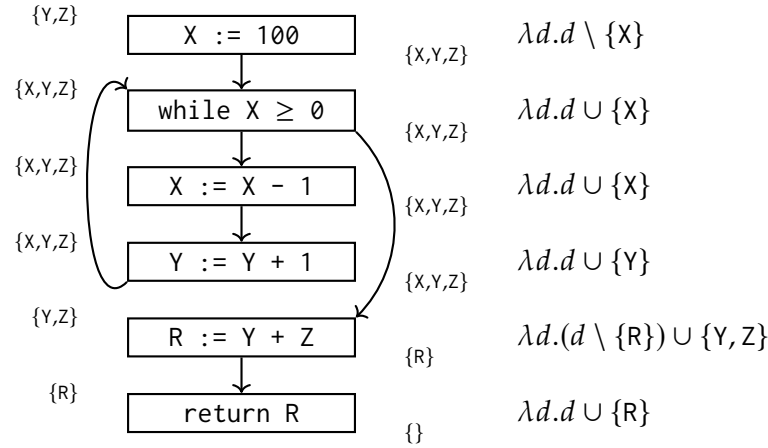


Figure 3.7. An imperative program and the result of live variable analysis. Data-flow values at every program point are shown around instruction nodes (incoming data-flow values to the left, data-flow values after an instruction to the right). To the far right are transfer functions associated with each instruction. The combined transfer function for the whole sequence of instructions is $\phi = \lambda d. (d \setminus \{X, R\}) \cup \{Y, Z\}$. The set of variables live at the start of the program can be computed by applying ϕ to the empty set: $\phi(\{\}) = \{Y, Z\}$.

analysis), and these constraints are of the form $\text{LiveIn}(n) = (\text{LiveOut}(n) \setminus \text{kill}(n)) \cup \text{gen}(n)$ for suitably defined functions kill and gen . Furthermore, the live-out set for each instruction n is a superset of the live-in sets for all of the instructions s that might be its successor (meaning there is an edge from n to s in the control-flow graph): $\text{LiveOut}(n) \supseteq \text{LiveIn}(s)$. This works in an imperative setting.

For all monotone data-flow analyses, and live variable analysis in particular, the notion of *transfer functions* ϕ (or *flow functions*) is as a way of representing the way data-flow values change in the program. For imperative liveness, the live-in variables for an instruction depend on the live-out variables, meaning that for each instruction n with transfer function ϕ_n the equality $\text{LiveIn}(n) = \phi_n(\text{LiveOut}(n))$ holds. The right part of Figure 3.7 shows the transfer functions for each instruction. For all monotone data-flow analyses, transfer functions are monotonic.

Transfer functions are also defined for blocks and sequences of instructions. For two instructions n_1 and n_2 , the live-in variables of n_1 depend on the live-out variables of n_2 by merely composing the transfer functions associated with n_1 and n_2 : $\text{LiveIn}(n_1) = \phi_{n_1}(\phi_{n_2}(\text{LiveOut}(n_2)))$. We then define the *transfer function for a computation* as a function mapping the variables live at computation end to the variables live at the start of the computation. We use this notion of transfer functions as the carrier set of the grading algebra that represents live variable analysis.

Given a fixed set of variables Variables , the grading algebra has as its carrier set all functions mapping sets of variables to sets of variables, which we write as $\mathcal{P}(\text{Variables}) \rightarrow \mathcal{P}(\text{Variables})$.⁴ Two functions f and g can be compared: we say $f \leq g$ if for all $S \subseteq \text{Variables}$, $f(S) \subseteq g(S)$. Sequencing is performed through function composition, as seen from the transfer function of $n_1; n_2$. The unit of function composition is just the identity id function of sets of variables. It is then

⁴We can be more precise and only consider *monotonic* endofunctions on $\mathcal{P}(\text{Variables})$, but this does not meaningfully change any of the following conclusions.

not difficult to show that all requirements are met for the following algebra to be a partially-ordered monoid:

$$(\mathcal{P}(\text{Variables}) \rightarrow \mathcal{P}(\text{Variables}), \leq, \circ, \text{id})$$

Grade as information

The simplest possible monad graded by the algebra of transfer functions is one which is just a variant of a **State** monad extended to have more than one component in which the grade does not affect the underlying data representation – it merely provides information to the programmer. We call this graded monad **LiveMultiState**_{triv}. I present this graded monad informally for the sake of motivation, and there is no translation to Target. This graded monad is used as a stepping stone to a more sophisticated **LiveMultiState** in which the grade *does* affect the representation of the computation. In turn, **LiveMultiState** has a simplified formulation in form of **LiveState**_S, which features again only one mutable variable, but is based on reasoning used for **LiveMultiState**.

Suppose there are n mutable variables in the *Variables* set, and we furthermore assume their types are all **Int** for simplicity. For any grade e , the type **LiveMultiState**_{triv} ^{e} A can be represented as $(\text{Int} \times \dots \times \text{Int}) \rightarrow (\langle\langle A \rangle\rangle_{\text{type}} \times (\text{Int} \times \dots \times \text{Int}))$, where there are n components in the $\text{Int} \times \dots \times \text{Int}$ tuples. The definition of this graded monad (meaning the bind and return) is exactly the same as that of the **State**_{Int $\times \dots \times$ Int} monad, and subeffecting/upcasting is trivial.

The main point of interest are the operations for reading from and writing to the state components in **LiveMultiState**_{triv}. Each mutable variable is a named member of *Variables*, and for all $X \in \text{Variables}$ there are two operations for interacting with this component of the store – get_X and put_X . Performing these computations respectively adds and removes X from the set of live variables, which means that the associated grades are transfer functions that add and remove X from a set:

$$\begin{aligned} \text{gen}_X &= \lambda d. d \cup \{X\} \\ \text{kill}_X &= \lambda d. d \setminus \{X\} \end{aligned}$$

The computational grading algebra used for representing live variable analysis is thus:

$$(\mathcal{P}(\text{Variables})^{\mathcal{P}(\text{Variables})}, \leq, \circ, \text{id}, \bigcup_{X \in \text{Variables}} \{\text{gen}_X, \text{kill}_X\})$$

For example, assuming $\text{Variables} = \{X, Y\}$, the following are the definitions and types of the stateful operations:

$$\begin{aligned} \text{get}_X &: \text{LiveMultiState}_{\text{triv}}^{\text{gen}_X} \text{Int} \\ \langle\langle \text{get}_X \rangle\rangle_{\text{val}} &= \text{fn } s : \text{Int} \times \text{Int} \Rightarrow \langle \text{proj}_1 s, s \rangle \\ \text{put}_X &: \text{Int} \rightarrow \text{LiveMultiState}_{\text{triv}}^{\text{kill}_X} \text{Unit} \\ \langle\langle \text{put}_X \rangle\rangle_{\text{val}} &= \text{fn } s : \text{Int} \Rightarrow \text{fn } s' : \text{Int} \times \text{Int} \Rightarrow \langle \langle \rangle, \langle s, \text{proj}_2 s' \rangle \rangle \\ \text{get}_Y &: \text{LiveMultiState}_{\text{triv}}^{\text{gen}_Y} \text{Int} \\ \langle\langle \text{get}_Y \rangle\rangle_{\text{val}} &= \text{fn } s : \text{Int} \times \text{Int} \Rightarrow \langle \text{proj}_2 s, s \rangle \\ \text{put}_Y &: \text{Int} \rightarrow \text{LiveMultiState}_{\text{triv}}^{\text{kill}_Y} \text{Unit} \\ \langle\langle \text{put}_Y \rangle\rangle_{\text{val}} &= \text{fn } s : \text{Int} \Rightarrow \text{fn } s' : \text{Int} \times \text{Int} \Rightarrow \langle \langle \rangle, \langle \text{proj}_2 s', s \rangle \rangle \end{aligned}$$

Analogous definitions hold for other sets of mutable variables.

Using transfer functions as grades in this way already provides the ability to perform live variable analysis. For example, let g be an expression of type $\text{LiveMultiState}_{\text{triv}}^\phi \text{Int}$. Consider a computation that reads from X and Y and stores the outcome in Z , after which it runs g :

$\text{bind } x : \text{Int} \leftarrow \text{get}_X \text{ with } (\text{bind } y : \text{Int} \leftarrow \text{get}_Y \text{ with } (\text{bind } _ : \text{Unit} \leftarrow \text{put}_Z (x + y) \text{ with } g))$

The grade associated with this expression is $\text{gen}_X \odot \text{gen}_Y \odot \text{kill}_Z \odot \phi$, which evaluates to $\lambda d. (\phi(d) \setminus \{Z\}) \cup \{X, Y\}$.

Non-trivially graded monad

Previously we constructed the graded monadic type $\text{LiveMultiState}_{\text{triv}}^\phi A$ as a simple wrapper over the usual state monad; grade ϕ was a transfer function but it had no semantic meaning: the grades were merely decorations on types and did not affect the operations. We can instead use grades to *refine* the types and operations of the usual state monad by the liveness information, so that graded monad operations actually depend on the grades. In this case, refinement means restricting stores to subsets of the variables involved in a program.

First, given any subset $V \subseteq \text{Variables}$, we define the V -refined store $\text{Store}(V)$ to be $V \rightarrow \text{Int}$, writing $\hat{0}$ for the only element of $\text{Store}(\emptyset)$. Note that $\text{Store}(\text{Variables})$ recovers (up to isomorphism) the previous $(\text{Int}, \dots, \text{Int})$ component in their entirety.

We now defined $\text{LiveMultiState}^\phi$ whose input and output stores are computed from ϕ :

$$\langle\langle \text{LiveMultiState}^\phi A \rangle\rangle_{\text{type}} = \text{Store}(\text{reads}(\phi)) \rightarrow \langle\langle A \rangle\rangle_{\text{type}} \times \text{Store}(\text{footprint}(\phi))$$

For the input store, $\text{reads}(\phi) = \phi(\emptyset)$ gives us the subset of variables which are live-in and thus potentially read by a computation of type $\text{LiveMultiState}^\phi A$. For the output store, $\text{footprint}(\phi) = \phi(\emptyset) \cup (\text{Variables} \setminus \phi(\text{Variables}))$ gives the *footprint* (borrowing terminology from separation logic [ORY01]) containing those variables read or written by this computation.⁵ For example, the computation:

$\text{do } \{ x \leftarrow \text{get}_X; y \leftarrow \text{get}_Y; \text{put}_Z (x + y) \}$

has grade $\phi = \text{gen}_X \odot \text{gen}_Y \odot \text{kill}_Z = \lambda d. (d \setminus \{Z\}) \cup \{X, Y\}$ and thus $\text{reads}(\phi) = \phi(\emptyset) = \{X, Y\}$ and $\text{footprint}(\phi) = \{X, Y, Z\}$.

We do not give a detailed definition of LiveMultiState in this thesis, as it is complex and the same functionality can be achieved using a simpler approach. Details can be found in my publication with Mycroft and Orchard. [IMO20]

Back to single mutable state

Live variable analysis is only practically used in the context of computations with multiple variables. However, it is conveniently an analysis in which reading from or writing to a variable X does not affect the presence of any other variable $Y \neq X$ in the grade. Therefore, the live variable analyses for each of the mutable variables can be seen as being done ‘independently’ or that the analysis is ‘separable’. It makes

⁵A more refined graded state monad would return an output store containing only those variables that are written to. However, liveness analysis alone does not allow us to compute just the variables written-to. The footprint is therefore a safe over-approximation of the written-to set.

gen	o	kill	id	gen
	kill	kill	kill	kill
id	id	kill	id	gen
	gen	gen	gen	gen
kill				

Figure 3.8. The computational grading algebra $(\{\text{kill}, \text{id}, \text{gen}\}, \sqsubseteq, \circ, \text{id}, \{\text{kill}, \text{gen}\})$ visualised.

sense to consider a simplified case of using only *one* mutable variable. Stateful computations with one mutable variable of type S can be thus represented using terms of a graded monad LiveState_S , where the grade is the transfer function of the computation and affect the data representation.

The algebra of transfer functions on a singleton set has only three elements: an identity function, removal of the variable from the live set and addition of the variable to the live set. These can be seen as three separate values id , kill and gen . The computational grading algebra can be succinctly defined (Figure 3.8).

In Source, the three state manipulation constructs are the same as in PermState_S , except their types differ:

$$\begin{aligned}
 (\text{GET}) \quad & \frac{}{\Gamma \vdash \text{get} : \text{LiveState}_S^{\text{gen}} S} \\
 (\text{PUT}) \quad & \frac{}{\Gamma \vdash \text{put} : S \rightarrow \text{LiveState}_S^{\text{kill}} \text{Unit}} \\
 (\text{RUN}) \quad & \frac{\Gamma \vdash M : \text{LiveState}_S^e A}{\Gamma \vdash \text{run } M : S \rightarrow A}
 \end{aligned}$$

The Source-to-Target translation is inspired by LiveMultiState : it is merely an instantiation for *one* state component, and thus a lot of the involved definitions simplify. The translation is deferred to the Appendix – more specifically, it is given in Figures E.1 and E.2.

In Chapter 7, when we introduce *graded monad transformers*, the graded monad LiveMultiState becomes unnecessary and can be represented in an analogous way to (non-graded) monadic multi-state computations: via a stack of graded monad transformers corresponding to the LiveState_S monads.

Haskell implementation

The $\text{LiveMultiState}_{\text{triv}}$ graded monad can be embedded into Haskell without significant effort, while the LiveMultiState embedding is more involved. The main challenge is the representation of the algebra of transfer functions, which uses type-level sets (from Orchard’s `type-level-sets` package) and an appropriate choice of type classes.

The core text of this thesis does not cover the details of this implementation. The main point important to notice is that each of the get_X , get_Y , get_Z are represented as separate graded monadic computations getX , getY , getZ , and similar for the `put` functions. The type of the `get` Haskell embeddings are as follows:

```

getX :: LiveMultiState (Gen "X") Int
getY :: LiveMultiState (Gen "Y") Int

```

```
getZ :: LiveMultiState (Gen "Z") Int
```

Note the use of strings in grades, which have to be stated explicitly and manually for each of the get functions. This link between the grade and the semantics has to be provided by the programmer who defines the graded monad operations.

Recursion The Haskell embedding works well for sequential code, but reaches its limits with branching and recursion as GHC is unable to compute fixed-points for the type equations generated for each of the mutually recursive graded monadic computations. Haskell does not support *equirecursive types*, and we pay special attention to this matter later.

The immediate effect of this fact is that the Haskell embedding, while it does translate the question of calculating live variables into type inference, the resulting type inference problem cannot be solved. Rather than inferring the grades, the programmer has to provide typing annotations. Type checking is then much easier to perform, all that needs to be done is that grade equalities need to be decided (which is easy in the case of liveness transfer functions because they are represented using get and put).

Monotone data-flow analysis in general

Live variable analysis is just one of several classical data-flow analyses that fall under the umbrella of monotone data-flow analysis, a class which includes reaching definitions, available expressions and similar program analyses. All monotone data-flow analyses have corresponding transfer functions, and these transfer functions form a monoid with a function composition operator.⁶ However, the data-flow values of these analyses are not simply sets of variables, as is the case with liveness. For example, reaching definitions data-flow values are sets of program labels corresponding to each assignment or write. Data-flow values are more complex for analyses such as constant propagation, available expressions and very busy expressions. In an imperative setting, the data-flow values and the transfer functions depend on assignments in their entirety. For example, data-flow values in constant propagation are maps from variables to constants (or $\top$ meaning ‘not a constant’, in addition to a $\perp$ data-flow value representing ‘no information’), so data-flow values associated with an assignment $X := E$ depend on the values of all variables that occur in E . This means that splitting assignments into appropriate get and put expressions, while resulting in semantically equivalent computations, cannot be assigned grades that represent the analysis.

The programmer can still represent the analysis by using more complicated graded monadic computations as building blocks for producing code and analyses. One way of doing this is by defining a separate graded monadic value for every assignment and providing a manual typing annotation that includes the corresponding transfer function grade. For example, assignments $\ell_1 : X := Y + Z$ and $\ell_2 : Z := X + 1$ are represented as two graded monadic computations `assign1` and `assign2`, and their definitions would mention the transfer functions associated with assignments $X := Y + Z$ and $Z := X + 1$.

⁶Forward analyses, such as constant propagation, have ‘reverse function composition’ as the appropriate operator: $\phi_1 \odot \phi_2 = \phi_2 \circ \phi_1$. This is just a matter of convention related to how function composition is written, we could write $\phi_1; \phi_2 = \phi_2 \circ \phi_1$.

This approach leaves much to be desired: specifically stating the type of every graded monadic assignment is cumbersome and not useful from a programming side – assuming a type system similar to that of Haskell. However, it still demonstrates that it is possible to incorporate these analyses into a graded monadic framework. More sophisticated methods of representing such analyses might exist, though investigating this question is outside the scope of this thesis.

3.3.3 Logging updates

The **PermState_S** graded monad states through its grade whether a computation writes to the underlying state. In addition to knowing whether any updates are made to the state, the programmer might be interested in *how many* are made, as well as the outcomes of all of the updates. It is possible to achieve this using a *history-graded* monad **HistState_S**, for S the underlying type of the store. This number of updates is then expressed through the grade in a graded monadic type of a computation – if a state gets updated at most three times, a representation of the number three is the grade.

History-graded monad

To make **HistState_S** an analysis-graded monad, its grading algebra needs to have a top element. A very simple grading algebra could distinguish between the cases ‘no updates’, ‘at most one update’ and ‘possibly more than one update’, but we can go beyond that. An analysis regarding the number of updates made to a state in a computation has two possible classes of outcomes:

- a natural number n as an upper bound on the number of updates;
- unbounded number of updates, an outcome which we represent as ω (possible if the computation is recursive, when there is no decidable bound on the number of state updates).

Each natural number bound n is subsumed by analysis outcome m for any $m > n$ and by ∞ . Thus, the upper bound on the number of updates made for a stateful computation is $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$, and these form a simple total order $(\mathbb{N}_\omega, \leq_\omega)$ for which $\leq_\omega$ is defined as follows:

$$\begin{array}{lll} m & \leq_\omega & n \quad \text{for } m, n \in \mathbb{N} \text{ and } m \leq n \\ m & \leq_\omega & \omega \quad \text{for all } m \in \mathbb{N}_\omega \end{array}$$

The order is visualised in Figure 3.9. Furthermore, we lift addition to work with ω using the following $+_\omega$ operator:

$$\begin{array}{lll} m & +_\omega & n = m + n \quad \text{for } m, n \in \mathbb{N} \\ \omega & +_\omega & e = \omega \quad \text{for all } e \in \mathbb{N}_\omega \\ e & +_\omega & \omega = \omega \quad \text{for all } e \in \mathbb{N}_\omega \end{array}$$

The grading algebra used for **HistState_S** is thus $(\mathbb{N}_\omega, \leq_\omega, +_\omega, 0, \{1, 0\})$. This is an example of an infinite grading algebra with an infinite ascending chain $(0 \leq_\omega 1 \leq_\omega 2 \leq_\omega \dots)$.

It remains to define the graded monad itself. In Source, we once again use two basic state manipulation computations (get and the put function), as well as a way

of running and getting the result of a computation:

$$\begin{aligned}
 (\text{GET}) \quad & \frac{}{\Gamma \vdash \text{get} : \text{HistState}_S^0 S} \\
 (\text{PUT}) \quad & \frac{}{\Gamma \vdash \text{put} : S \rightarrow \text{HistState}_S^1 \text{Unit}} \\
 (\text{RUN}) \quad & \frac{\Gamma \vdash M : \text{HistState}_S^e A}{\Gamma \vdash \text{run } M : S \rightarrow A}
 \end{aligned}$$

It is not directly possible, using the core language as presented, to devise a computation whose type is $\text{HistState}_S^\omega A$ for any S and A . However, the addition of recursion allows us to write programs with a potentially unbounded number of state updates.

The translation to Target requires an additional data-type representing lists. The main idea is that, for any grade $e \in \mathbb{N}_\omega$, the representation of graded monadic types should be

$$\text{HistState}_S^e A = S \rightarrow (A \times S^{\leq_\omega e + \omega 1})$$

where we use the notation $S^{\leq_\omega F}$ to represent tuples that have up to F components:

$$S^{\leq_\omega n + \omega 1} = \bigcup_{k=1}^{n + \omega 1} S^k \qquad S^{\leq_\omega \omega} = \bigcup_{k=1}^{\infty} S^k$$

In the case of $\text{HistState}_S^\omega A$, there is a (finite) list of intermediate values of the state along with the result. We choose to list the most up-to-date values first – if the state was initially 5, and the state after two updates was 1, and then 7, the tuple representation of this history is $\langle 7, 1, 5 \rangle$.

I do not provide a full translation to Target, instead I sketch the main ideas (see Haskell implementation in Appendix C for details). Subeffecting coercion is trivial, as $S^{\leq_\omega e + \omega 1} \subseteq S^{\leq_\omega f + \omega 1}$ for all $e \leq_\omega f$. Returning is a variant of the usual `return` for the State_S monad, except the state is wrapped inside a one-element tuple:

$$\langle \langle \Gamma \vdash \text{return } M : \text{HistState}_S^0 A \rangle \rangle_{\text{val}} = \text{fn } s : \langle \langle S \rangle \rangle_{\text{type}} \Rightarrow \langle \langle \Gamma \vdash M : A \rangle \rangle_{\text{val}}, \langle s \rangle \rangle$$

Graded bind is not difficult: we list all of the intermediate values of the state from both computations. It is important to not list the final state of the first computation twice – it is the same as the initial state of the second computation: $\langle 2, 4 \rangle$ and $\langle 7, 9, 2 \rangle$ are plausible histories for two computations being sequenced, and the result of merging the histories should be $\langle 7, 9, 2, 4 \rangle$. Aside from this fact, it is a simple matter of appending two tuples:

$$\begin{aligned}
 \langle \langle \Gamma \vdash \text{bind } x : A \leftarrow M \text{ with } N : \text{HistState}_S^{e + \omega f} \rangle \rangle_{\text{val}} = \\
 \text{fn } s : \langle \langle S \rangle \rangle_{\text{type}} \Rightarrow \text{let } \langle r, h_1 \rangle = t_1 \text{ } s \text{ in} \\
 \text{let } \langle y, h_2 \rangle = (t_2 \{r/x\}) \text{ (hd } h_1) \text{ in} \\
 \langle y, \text{append } h_2 \text{ (tl } h_2) \rangle
 \end{aligned}$$

It can be shown that this definition satisfies all properties expected of a graded monad.



Figure 3.9. The $(\mathbb{N}_\omega, \leq_\omega)$ total order.

Haskell embedding

The `HistStates` graded monad can be embedded into Haskell without much trouble, and is featured in the `analysis-monad` library.

When it comes to representing the grading algebra, the key idea is that $\mathbb{N}_\omega$ is represented as a type-level sum `NatInf`: it is either a Peano-encoded type-level natural number or ω , represented by type `Inf`. Addition $+_\omega$ is easily encoded using a type family `AddInf`.

The underlying representation of `HistStatese A` requires devising a representation for $S^{\leq_\omega e + \omega^1}$. For any $e \in \mathbb{N}_\omega$, the type of lists whose length is exactly e if $e \in \mathbb{N}$ is represented using a GADT `Vec n a`:

```
data Vec (n :: Nat) a where
  VNil  :: Vec Zero a
  (:::) :: {vHead :: a, vTail :: Vec n a} -> Vec (Succ n) a
```

Representing lists whose length is at most n uses a further GADT. For example, a tuple whose length is at most $n + 1$ is either a list whose length is at most n or a vector whose length is exactly $n + 1$. Special treatment is given to ω : any finite list has length ‘at most ω ’. The definition of this representation is thus:

```
data VecTo (n :: Nat) a where
  FromVec :: Vec n a -> VecTo n a
  Ext      :: {unExt :: VecTo n a} -> VecTo (Succ n) a
data HistState s (f :: NatInf) a where
  FinHistState :: {runFinHistState :: s -> (a, VecTo n s)}
                 -> HistState s (Fin n) a
  InfHistState :: {runInfHistState :: s -> (a, [s])}
                 -> HistState s Inf a
```

The remaining parts of the translation are straightforward and simpler than for permissions.

Related work

Graded monads representing stateful computations whose grade informs about the number of updates have been given as examples in the past – for example Fujii et al. [FKM16] use it as a running example in their formal 2-categorical treatment of graded monads. Their example considers a simpler grading algebra than the one present here, namely just the preordered monoid of natural numbers with addition – excluding a special top ‘infinity’ symbol. Excluding ω means that the graded

monad does not represent failure to find a bound as a possible outcome of the analysis. Uustalu [Uus20] uses a similar grading algebra, again without ω , in the context of a graded variant of the **Writer** monad. The **HistState**_s monad thus gives a more complete view of counting updates than these previous attempts.

Relation to linearity

A related question to counting the number of updates to a store is counting *usage*, which is typically used to reason about resources. Facts about resource usage, for example whether a particular value is used at most once in an expression, is handled by variants of *linear type systems*. [Laf88] Basic linearly-typed languages qualify each variable as either linear or unrestricted, meaning they are respectively either used at most once⁷ or there is no such restriction on the variable. This is important in effectful languages in which we wish to guarantee that some data is not unsafely cloned (‘leaked’) to another part of the program, or that a file is not opened more than once. Such type systems need to be able to handle `let  $x = M$  in  $N$` expressions as well: the variable y in `let  $x = y$  in  $\langle x, x \rangle$` really gets used twice, so it cannot be linear. This idea can be extended to more general *usage analysis* in which each variable has an associated number (potentially ω) of possible reads.

Both of these type systems are examples of *structural coefficient systems* described by Petriček [POM14, Pet17], where the context of a typing judgement is extended with additional contextual information about each variable. Structural type-and-coeffect judgements are of the form $\Gamma @ R \vdash M : A$, which are similar to the familiar $\Gamma \vdash M : A$ typing judgements, along with a vector R of the same length as Γ in which for every i , the i -th component of R is a *coeffect* value associated with the i -th variable in Γ . The number of uses of a variable is an example of a coeffect. Thus, the judgement $x : \text{Int}, y : \text{Int} @ \langle 2, 3 \rangle \vdash M : A$ states that x is used at most 2 times in M , and y is used at most 3 times.

A feature of coeffects is the existence of two ‘base’ coeffect values: one represents an ignored variable (thus sometimes denoted by **ign**) and another represents the use of a variable (usually denoted **use**). For usage analysis, these are 0 and 1, respectively.

These facts already make it difficult to represent usage analysis using effects and graded monads. Binding a ‘get’ computation with a function `fn  $x \Rightarrow \text{return } \langle x, x \rangle$` means the underlying state is used twice, but the grade associated with `return  $\langle x, x \rangle$` is the identity of an algebra representing usage analysis, not 2. Usage analysis is not modelled by techniques of this thesis.

3.4 Safe concurrency

Concurrent programs with shared memory are notoriously difficult to reason about and write correctly. There are various failure modes for concurrent programs, most prominently *race conditions* (parallel threads read from and write to a shared state in a way that makes the program state inconsistent) and *deadlock* (a set of threads are waiting for each other to finish in order to proceed). There are a variety of static and runtime analysis tools that can be used to aid programmers in finding and fixing concurrency bugs.

⁷in the sense of semantics, not syntactically: in `if  $b$  then  $x + 1$  else  $x + 2$` the variable x is used once whatever the value of b is, but it appears twice in the expression

It turns out that many static safety properties can be inferred and reasoned about using graded monads. This section is dedicated to several graded monads used for reasoning about the safety of concurrent computation. These concern the problem of deadlock and their goal is to establish whether a particular lock usage order is respected by the individual threads. The first one is simple and merely checks whether a given predetermined order is respected. The second one is meant to be used in a setting where a decidable grade inference algorithm exists, and it is then capable of *inferring* the order in which locks are acquired and released. A common feature of all of these graded monads for safe locking is that the underlying data representation does not depend on the grade, so the grade only tells the programmer the outcome of the analysis (as opposed to `PermStateS`).

For these analysis-graded monads we do *not* consider a translation to the Target language. Instead, the focus is on Haskell. This is because Target is a pure functional language with no capabilities for performing I/O, so adding concurrency would complicate the presentation.

3.4.1 Avoiding deadlock with structured locking

One of the simplest ways of performing synchronisation between concurrent threads is by using mutual exclusion (mutex) locks. A thread can either acquire or release a lock. If a thread t_1 attempts to acquire a lock ℓ but another thread t_2 holds ℓ , then t_1 blocks – and when t_2 releases ℓ , a thread blocking on ℓ gets woken up. A common locking discipline obeyed by concurrent programs is that lock acquisition and release is *structured*: if a thread acquires lock ℓ_1 before acquiring lock ℓ_2 , then it must release ℓ_2 before releasing ℓ_1 .

If all threads use a structured locking discipline, and furthermore lock acquisition obeys a total order (ℓ_1 always acquired before ℓ_2 , which is always acquired before $\ell_3 \dots$), then it is guaranteed that the program cannot deadlock [Dijnd]. We use this fact to design a graded monad and an appropriate grading algebra representing concurrent computation and reason about its safety. There are two main constructs for programming with concurrent programs in this framework:

Parallelism Given two computations M and N representing threads, there is a computation $M \parallel N$ representing a computation running M and N in parallel.

Synchronisation Given a threaded computation M and a lock ℓ , the computation `sync  $\ell$   $M$` is a thread that first acquires ℓ , then runs M , and finally releases ℓ .

The graded monad `Sync` represents computations built up using these parallelism and synchronisation, meaning that graded monadic types `Sync $e$   $A$` are the types of these computations. Since these are impure, I/O computations, the grades e are merely descriptive and give the outcome of an analysis, as opposed to changing the underlying data representation.

Given a set of locks $\mathcal{L} = \{\ell_1, \ell_2, \dots, \ell_k\}$, ordered as $\ell_1 < \ell_2 < \dots < \ell_k$, the grading algebra is the algebra whose carrier set elements are either $\top$, meaning ‘order is not obeyed’, or a set $S \subseteq \mathcal{L}$ representing the set of all locks acquired (and released) by the computation. The partial order and the sequencing operator are just subset and union, taken from a powerset algebra, extended to work with a top element $\top$: $e \subseteq \top$ and $e \cup_{\top} \top = \top \cup_{\top} e = \top$ for any $e \in \mathcal{P}(\mathcal{L}) \cup \{\top\}$. The algebra is thus:

$$(\mathcal{P}(\mathcal{L}) \cup \{\top\}, \subseteq_{\top}, \cup_{\top}, \emptyset, \{\cup_{\top}\} \cup \{\text{pcons}_{\ell}\}_{\ell \in \mathcal{L}})$$

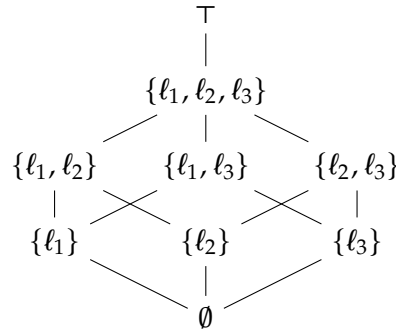


Figure 3.10. The locking grading algebra join-semillattice for $\mathcal{L} = \{\ell_1, \ell_2, \ell_3\}$.

The poset is visualised in Figure 3.10.

In the language we introduced, the typing rules for synchronisation and parallelism are as follows:

$$\begin{aligned}
 (\text{PAR}) \quad & \frac{\Gamma \vdash M : \text{Sync}^e A \quad \Gamma \vdash N : \text{Sync}^f B \quad \vdash g = e \cup_{\top} f}{\Gamma \vdash M \parallel N : \text{Sync}^g (A \times B)} \\
 (\text{SYNC}) \quad & \frac{\Gamma \vdash M : \text{Sync}^e A \quad \vdash f = \text{pcons}_{\ell}(e)}{\Gamma \vdash \text{sync } \ell \ M : \text{Sync}^f A}
 \end{aligned}$$

The rule for parallel computation is simple: the set of locks acquired by $M \parallel N$ is the union (with $\top$ accounted for) of the locks in the individual threads. The rule for synchronisation is more sophisticated and makes use of the ‘partial add’ function pcons , which for a given lock ℓ and grade F produces a grade representing the fact that a lock ℓ is acquired before any of the locks represented by grade e . This function is formally defined as follows:

$$\begin{aligned}
 \text{pcons}_{\ell}(\top) &= \top \\
 \text{pcons}_{\ell}(S) &= \begin{cases} \{\ell\} \cup S, & \forall \ell' \in S. \ell < \ell' \\ \top, & \text{otherwise} \end{cases}
 \end{aligned}$$

This means that the only way $\text{pcons}_{\ell}(e)$ is not $\top$ is when e is a set of locks and ℓ comes before all of the locks in this set.

Haskell embedding

The Sync graded monad can be represented in Haskell without much difficulty. Subsets of the set of locks are represented as type-level lists of strings representing lock names. Parallelism is achieved using a $\langle | | \rangle$ operator, and synchronisation for each lock is represented using a separate graded monadic function, for example syncX , syncY , syncZ . Implementing these operators uses low-level concurrency constructs in Haskell, in particular MVar for synchronisation and forkIO for spawning new threads (needed for implementing $\langle | | \rangle$). As usual, all graded monad operators are represented using type families.

The underlying representation of $\text{Sync } f \ a$ is as an **IO** computation utilising a fixed number of MVars ⁸ representing the individual mutual exclusion locks. Accessing the lock reference is a read-only operation (the reference cannot be changed),

⁸data protected by a mutual exclusion lock in Haskell, frequently used as a building block for locking mechanisms

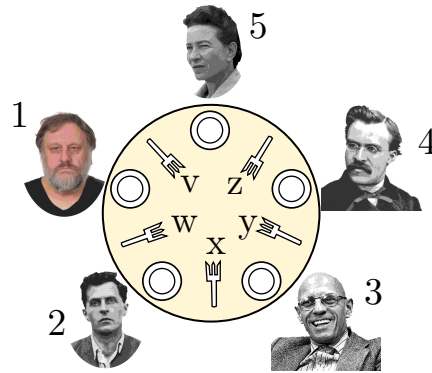


Figure 3.11. Philosophers enjoying their dinner: (1) Slavoj Žižek; (2) Ludwig Wittgenstein; (3) Michel Foucault; (4) Friedrich Nietzsche; (5) Simone de Beauvoir.

meaning that we leverage the `ReaderT` monad transformer in which the underlying state is a mutex lock reference. For example, a concurrent computation using at most two locks can be represented in the following way:

```
data Sync f a = ReaderT (MVar ()) (ReaderT (MVar ()) (IO a))
```

Practical programming in this concurrent setting requires making **IO** computations concurrent. Hence `Sync`-typed computations `delay` and `syncPutStrLn` are introduced, representing waiting a given number of microseconds and printing to the standard output.

Example 3.1. The dining philosophers problem is a commonly used example of concurrent programming where deadlock may occur (Figure 3.11). It can be phrased in the following way:

Five philosophers are sitting at a round table and eating spaghetti for dinner. Their dexterity leaves a lot to be desired, for they are easily distracted and any attempt at eating with only one fork will result in a mess. Hence each philosopher will need two forks. There are exactly five forks on the table, one between each pair of adjacent philosophers. The philosophers must remain seated for the duration of the dinner and all of them must eventually finish eating.

A simple solution immediately comes to mind: each philosopher first acquires the fork to their left, then the fork to their right, then they ‘eat’. Unfortunately, this solution runs the risk of leading to deadlock. Instead, each philosopher acquires the lock to the left or right that comes earlier in the order of locks, and then the lock coming later in the order (in other words, first acquire the minimum of the two locks, then the maximum).

The embedding of this framework into Haskell illustrates this easily. Firstly, the logic for a particular philosopher `x`, with forks being represented as synchronisation functions `s1` and `s2`, is given by the following computation:

```
philosopher x s1 s2 = sync s1 $ do
  delay 100000
  sync s2 $ do
```

```

delay 100000
syncPutStrLn $ "philosopher_" ++ show x

```

We can define five philosopher behaviours by giving appropriate parameters to philosopher:

```

philosopher1 = philosopher 1 syncV syncW
philosopher2 = philosopher 2 syncW syncX
philosopher3 = philosopher 3 syncX syncY
philosopher4 = philosopher 4 syncY syncZ
philosopher5 = philosopher 5 syncZ syncV
philosopher5' = philosopher 5 syncV syncZ

```

The first solution is represented by the computation:

```

philosopher1 || philosopher2 || philosopher3
|| philosopher4 || philosopher5

```

and its grade is $\top$, illustrating the fact that the lock ordering is not obeyed. The second solution has a type-level list as its grade (a list of symbols "V", "W", "X", "Y", "Z"), meaning that the computation is free from deadlock:

```

philosopher1 || philosopher2 || philosopher3
|| philosopher4 || philosopher5'

```

The Haskell embedding can be easily extended to support reasoning about and programming with memory regions: rather than a lock being a mutex lock on its own, it is associated with some shared memory. This is achieved by changing the read-only state in the reader monad: for example, rather than `ReaderT (MVar ())`, a lock on a memory region containing an integer and a string is introduced with a monad transformer `ReaderT (MVar (Int, String))`. Details are not provided in this thesis, but an interested reader can read about this embedding in the associated functional pearl [IM20].

3.4.2 Graph-based analysis

The Sync graded monad provides a neat way of writing concurrent programs based on structured locking. The major practical downside of this graded monad is the fact that it requires the programmer to explicitly state the lock total order, as opposed to having that aspect inferred by the type system. However, this additional work can be avoided by using a different grading algebra. The grade of a computation M should instead state not only which locks are acquired by M , but also the *inferred partial order* for these locks – if any compatible ordering exists.

Consider computations using mutual exclusion locks drawn from $\mathcal{L}$. The elements of this algebra are either $\top$ – representing a potentially unsafe computation – or pairs $\langle L, R \rangle$, where $L \subseteq \mathcal{L}$ is a set of locks acquired by a computation and $R \subseteq L \times L$ is as general as possible irreflexive order of locks that the computation obeys. For example, a computation acquiring and releasing ℓ_1 prior to locking ℓ_2 , then locking and releasing ℓ_3 , and finally releasing ℓ_2 , has $\langle \{\ell_1, \ell_2, \ell_3\}, \{\langle \ell_2, \ell_3 \rangle\} \rangle$ as the associated grade. These pairs $\langle L, R \rangle$ can also be seen as graphs: L is the set of vertices and R is the set of edges. Since the relation R is an irreflexive ordering, it is transitively closed and contains no cycles.

The ordering $\sqsubseteq$ on these grades is simple: $e \sqsubseteq \top$ for any F , and $\langle L_1, R_1 \rangle \sqsubseteq \langle L_2, R_2 \rangle$ if and only if $L_1 \subseteq L_2$ and $R_1 \subseteq R_2$

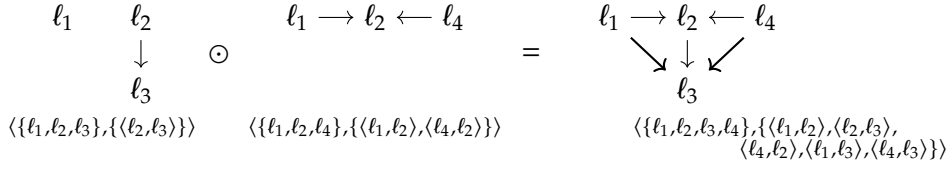


Figure 3.12. Visualising sequencing with graphs. The thicker edges are the ones added in order to make the graph represent a partial order (added by the transitive closure operator).

Running two computations wrapped inside **Sync** sequentially or in parallel combines the underlying sets of locks. In addition, the two lock orders have to be checked for compatibility: is there a common partial order for both of the computations being sequenced? Suppose R_1 and R_2 are the two lock orders. The union of these two relations (in graph terms, sets of edges) is not necessarily transitive, so we consider the transitive closure of $R_1 \cup R_2$, which we denote $(R_1 \cup R_2)^*$. If this transitive closure is irreflexive, then $R_1 \cup R_2$ contains no cycles and the combined computation is safe. Otherwise, the combined computation is unsafe, and the result of sequencing the two grades is $\top$. Therefore, the sequencing operator of this algebra is defined as follows:

$$\begin{array}{llll}
e & \odot & \top & = \top \\
\top & \odot & e & = \top \\
\langle L_1, R_1 \rangle & \odot & \langle L_2, R_2 \rangle & = \begin{cases} \top, & R_1 \cup R_2 \text{ contains a cycle} \\ \langle L_1 \cup L_2, (R_1 \cup R_2)^* \rangle, & \text{otherwise} \end{cases}
\end{array}$$

An example of how two such grades are sequenced, using the graph interpretation of the carrier set, is given in Figure 3.12.

This completes the definition of the grading algebra:

$$(\{\top\} \cup (\mathcal{P}(\mathcal{L}) \times \mathcal{P}(\mathcal{L} \times \mathcal{L})), \sqsubseteq, \odot, \langle \emptyset, \emptyset \rangle, \{\sqcup\} \cup \{\text{pcons}_\ell\}_{\ell \in \mathcal{L}})$$

Synchronisation is a different matter and we need to enhance the definition of **pcons**. Synchronising a computation M on lock ℓ is safe if and only if ℓ is not used by M . Otherwise ℓ comes before any of the locks used by M in **sync** ℓ M . Hence the definition of **pcons** $_\ell$ is now:

$$\begin{array}{ll}
\text{pcons}_\ell(\top) & = \top \\
\text{pcons}_\ell(\langle L, R \rangle) & = \begin{cases} \top, & \ell \in L \\ \langle L \cup \{\ell\}, R \cup \{\langle \ell, \ell' \rangle \mid \ell' \in L\} \rangle, & \ell \notin L \end{cases}
\end{array}$$

With $\odot$ and **pcons** defined, we can update the typing rules for parallelism and synchronisation:

$$\begin{array}{c}
(\text{PAR}) \quad \frac{\Gamma \vdash M : \text{Sync}^e A \quad \Gamma \vdash N : \text{Sync}^f B \quad \vdash e \odot f = h}{\Gamma \vdash M \parallel N : \text{Sync}^h(A \times B)} \\
(\text{SYNC}) \quad \frac{\Gamma \vdash M : \text{Sync}^e A \quad \vdash \text{pcons}_\ell(e) = f}{\Gamma \vdash \text{sync } \ell \text{ } M : \text{Sync}^f A}
\end{array}$$

Limits and generalisation This algebra does not force the programmer to specify the lock order ahead of time. However, there is an assumption that the lock order is constant throughout a computation. This is a restrictive assumption and it rejects reasonable programs in which threads that are never runs concurrently assume different lock orders. Consider the following computations:

$$\begin{aligned} M &= \text{sync } \ell_1 (\text{sync } \ell_2 (\text{return } 1)) \\ N &= \text{sync } \ell_2 (\text{sync } \ell_1 (\text{return } 2)) \end{aligned}$$

Running M and N in parallel is unsafe and might lead to deadlock. However, running them in sequence is safe, as all of the locks acquired by one computation are released prior to another acquiring them.

The sequencing operator can be thus modified to account for these cases. Rather than computing the transitive closure, sequencing two lock orders only returns their union. This means that for some locks x and y , both $\langle x, y \rangle$ and $\langle y, x \rangle$ can be listed in the lock ordering, but these hold in different parts of the computation. The grade of a parallel computation has to be updated accordingly. Running two safe computations in parallel is safe if the combined lock order has no cycle with at least one edge from *each* of the computations – otherwise it is unsafe.

Haskell embedding

Representing this variant of the **Sync** graded monad in Haskell has as its main difficulty the representation of the grading algebra, in particular its operators. Lock orders $\langle L, R \rangle$ are represented as type-level pairs of lists and lists of pairs. The most complicated part of the computation is computing the transitive closure of a relation, which is needed for sequencing. The source of the difficulty comes from the fact that this operator is required to be implemented using type families in Haskell. Type families in Haskell need to be written in such a way that the type system can statically infer that the computation terminates. This is desirable to avoid type checking looping at compile time.

The facilities for inferring that a type-level operator terminates are necessarily limited, since the question is undecidable. As a result, type-level conditionals require that *both* branches lead to termination: `If b Then p Else q` is checked for termination at compile time by checking whether both p and q terminate, even if one of the branches is impossible. This leads to issues with algorithms based on iteration until a stopping condition is met, such as ‘all vertices are visited’, and most graph algorithms are like this. The problem is resolved by writing every operator in a *primitive recursive* fashion, meaning that one of the arguments must reduce in every recursive call.

This is best seen in the type-level family which computes all the vertices reachable from a particular vertex in a graph. This type family `Reachable (v, e) src`, where (v, e) is the graph (with e an adjacency list) and src is the source vertex, relies on depth-first search implemented using an auxiliary type family with four arguments `Reachable' e pend vis lim`, in which `pend` is the stack of vertices that are scheduled to be expanded, `vis` is a list of vertices that have already been visited, a `lim` is an argument that decreases in every recursive call in order to provide the underlying depth-first search algorithm with a guarantee of termination (initially set to the number of vertices) to the Haskell type checker.

```
type family Reachable :: (k, [(k, k)]) -> k -> [k] where
```

```

Reachable (v, e) src = Reachable' e '[src] '[] (Length v)
type family Reachable' :: [(k, k)] -> [k] -> [k] -> Nat -> [k]
Reachable' _ _ vis 0 = vis
Reachable' _ '[] vis _ = vis
Reachable' e (p ': pend) vis n = If (Mem p vis)
  (Reachable' e pend vis (n - 1))
  (Reachable' e (Append (GetNeighbours e p) pend)
    (p ': vis) (n - 1))

```

3.4.3 Related work

Communication through channels

A way of modelling concurrency is through threads that synchronise and communicate using *message passing*. These threads are able to read and write data from and to *channels*: a thread t reading from a channel C means it awaits for some other thread to write to C , after which t is able to use this value. Message passing is in practice used in the Actor model of concurrency [HBS73], as well as in process calculi such as CSP [Hoa78] and π -calculus. [MPW92]

Session types [Hon93] are a formalism for modelling structured communication-based programming that provides computational guarantees. The simplest variant of session types enables exactly two processes to each use a particular channel, and their behaviour must be *dual*: each write from one thread exactly corresponds to a read from the other thread. This corresponds to a client-server model of computation in which no process infinitely blocks. Programs that do not satisfy this property do not type check in a type system with sessions.

Session types via graded monads Orchard and Yoshida [OY16] show that session types can be phrased using the language of effects and provide a Haskell embedding using graded monads (in the style of effect-monad). The main idea is that a process resulting in an value of type A is represented as a graded monadic type $\text{Process}^S A$, where S is the session type of the process. This allows the programmer to use graded monadic notation that more closely resembles the way process communication is usually written than the π -calculus syntax. For example, a server receiving a value via channel c and then sending its successor, has a simple graded monadic implementation:

```
do { x ← receive; send (x + 1) }
```

Graphs as grades

The idea of type-level graphs for managing relations and dependencies in a graded setting is not unique to this thesis. While I was writing this thesis, Keating and Gale [KG21] published a paper on how graded monads and type-level programming can be used for dependence analysis – specifically, the order of accessing memory locations by a stateful computation. This can then be used for establishing that two threads can be safely run concurrently, without any data races.

Keating and Gale’s work has a similar approach to type-level graph algorithms as this thesis, though arguably ‘more elegant’: instead of using additional ‘counter’ arguments in type families to guarantee termination, they use first-class type fam-

ilies (the first-**class**-families Haskell package) ⁹ and the ‘decreasing’ argument is a pair of a stack of to-visit and visited vertices.

3.5 Discussion and limitations

We now briefly turn to the consequences of the decisions made in the definition of analysis-graded monads. The main topic of interest are the various limitations and challenges when it comes to practical programming with them.

The question of distributivity

Consider a computation of the form:

$$M_1 \gg= (\text{fn } x \Rightarrow \text{alt } B \text{ then } M_2 \text{ else } M_3)$$

where **alt** is the ‘graded conditional’ (as seen in the Haskell embedding). If the types of M_1 , M_2 , M_3 are respectively $T^e A$, $T^{f_1} B$ and $T^{f_2} B$, following the typing derivation directly we infer that the type of the above expression is $T^{e \odot (f_1 \vee f_2)} B$. Dynamically the expression either enters one branch or the other, so the outcome of the analysis is more precisely described by $(e \odot f_1) \vee (e \odot f_2)$. In all preordered monoids the inequality $(e \odot f_1) \vee (e \odot f_2) \leq e \odot (f_1 \vee f_2)$ for all $e, f_1, f_2 \in E$ (by definition of $\vee$ – least upper bound). By analogy with the language of program analysis, we call computational grading algebras for which the inequality is actually an equality *distributive*. Formally, a computational grading algebra $(E, \leq, \odot, 1, O)$ is distributive if for all $e, f, g \in E$ the following equalities hold:

$$\begin{aligned} (e \vee f) \odot g &= (e \odot g) \vee (f \odot g) \\ e \odot (f \vee g) &= (e \odot f) \vee (e \odot g) \end{aligned}$$

Such properties are useful for a wide variety of program analyses. Popular general frameworks for performing data-flow analysis typically assume distributivity, for example IFDS. [RHS95] Gordon’s work on effect systems [Gor17, Gor18, Gor21] presupposes a distributivity axiom as it covers a vast range of program analyses.

There are examples of non-distributive analyses, of which *constant propagation* is the simplest one. This causes a problem for algebraic approaches to performing safe program transformations: they should either preserve the grade of the expression or overapproximate it. In implementation, constant propagation is sometimes restricted to merely performing *copy propagation*: a variable is considered constant if it is assigned a constant value or it copies the value of a variable marked as constant. Copy propagation and similar restricted variants of constant propagation are distributive.

The question of performing program optimisation guided by the grade of an expression is an interesting one, yet outside the scope of this thesis. I do not assume distributivity of computational grading algebras going forward.

Polymorphism

The Source language requires all function abstractions and graded monadic binds to have explicit typing annotations in order to make types unique for Source terms.

⁹<https://hackage.haskell.org/package/first-class-families-0.8.0.1>

This also means that the language features no parametric polymorphism (à la Haskell or ML). Parametric polymorphism is overall a useful feature to have in a language, but it comes with various well-known difficulties. We are particularly interested in *grade polymorphism* – for example, a term could have a polymorphic type $\forall_{\text{eff}} r (\text{PermState}_S^r \text{Int} \rightarrow \text{PermState}_S^{r \odot \text{ro}} \text{Int})$, meaning that it is a function that could be applied to any stateful computation.

Adding polymorphism to Source is addressed in Chapter 4, in particular for the sake of reinforcing the connection between grades and analysis outcomes (in other words, performing an analysis is equivalent to inferring the grade within monadic types).

Multi-component graded monads

The exception, multi-state liveness and locking analysis-graded monads (Sections 3.2.2, 3.3.2 and 3.4) have a *parameterised* set of graded monadic operations: each exception ϵ has a corresponding raise_ϵ , each mutable variable X has a corresponding get_X . This means that the grading algebra and the graded monadic operations depends on the underlying set of ‘building blocks’ (such as exception or variable names). The individual raise_ϵ and get_X operations are defined analogously, but translation into Target or a Haskell embedding does not allow a ‘generic’ way of defining these operations. A further challenge appears in the fact that the grading algebra representation (for example, of the operators in $\mathcal{O}$ or the sequencing $\odot$) requires sophisticated type-level programming facilities in Haskell.

We turn to this issue in more detail in Chapter 6 by introducing the notion of *graded monad builder*: in short, these are a way of defining an analysis-graded monad with a particular set of *symbols* (representing exception and variable names), which additionally provides all of the necessary boilerplate definitions that make programming with analysis-graded monads convenient.

3.6 Summary

In this chapter, I defined analysis-graded monads as a programming construct and a special case of general graded monads. They are capable of representing a variety of program properties and analyses related to impure concepts such as exceptions and stateful computation.

The contribution of this chapter is a terminology for discussing analysis-graded monads and a framework used in the following chapters (using Source and Target languages), as well as the examples that go beyond prior attempts to use graded monads for this purpose. All of the examples (except concurrency) are phrased in terms of a source language (Source) with analysis-graded monadic types, and a target language (Target), which is purely functional and into which all graded monadic operations and types are translated. In addition to the theory, the examples listed in the chapter are implemented in a new Haskell library called `analysis-monad`.

Grade inference

In Chapter 3 I defined analysis-graded monads and introduced the Source and Target languages. While the inspiration is program analysis and the theory of abstract interpretation, no algorithm is given for inferring the outcomes of analyses (meaning grades in graded monadic types). This is because Source requires all grades to be explicit, including in $\text{sub}_{e,f}$, and there is no recursion. A programmer would rather not perform the analysis herself – instead, grades should be computed using a *grade inference* algorithm. This is similar to type inference and also what one typically expects when the term ‘program analysis’ is used.

In this short chapter, we investigate the problem of inferring grades of graded monadic computations.

Chapter structure In Section 4.1, we first slightly modify Source to make grades unknown and formulate a grade inference problem. We then give an algorithm that computes the grade of the term in question. We tackle the challenge of adding *recursion* to the language, as the simple initial grade inference algorithm is not guaranteed to terminate in this case. We also consider the related problem of performing type inference in a variant of Source which supports parametric polymorphism for grades in Section 4.2. This leads to a more precise class of grading algebras which allow full grade inference, even in recursive programs, as well as a grade inference algorithm for such grading algebras.

4.1 Grade holes

A common way type inference is phrased in literature is by providing two languages: a typed one and one in which types are either entirely missing or erased. For example, Hindley-Milner type inference [Hin69, DM82] is commonly presented using two minimal languages, and the key result is that the inferred types in the unannotated language give a valid typing in the language in which all types are explicit. The problem considered here is mostly similar, though we limit our attention to inferring grades. This means that type annotations are still given where needed, such as in function abstractions, but grades are *not* stated. Instead, grades are replaced with *grade holes*, which then need to be computed and *filled* by a type inference algorithm to produce a valid program. For example, rather than writing the type $T^e \text{Int}$, the programmer writes T^{Int} .

Grade inference requires a notion of a *best* solution, meaning that among valid grades for a grade inference problem there needs to be a least one. As a result, we need to place some additional requirements on grading algebras and introduce changes to the Target programming language.

In this section, we first define Source° , a variant of Source with grade holes. We go on to elaborate on the necessary additions to grading algebras and analysis-graded monads. We then use Source° as a setting for a type inference algorithm and show that this algorithm returns a valid Source° typing.

4.1.1 A gradeless Source variant

The Source° language is defined as a variant of Source in which terms that require explicit type annotations have grades replaced by holes, written $\circ$. This is most relevant for writing functions and binds, for example in the case of the following term:

$\text{AddOne} = \text{fn } m : \text{GMaybe}^\circ \text{Int} \Rightarrow (\text{bind } x : \text{Int} \Leftarrow m \text{ with return } (x + 1))$

Here the programmer does not need to state explicitly whether the input computation represented by m is failing, successful or whether this question is statically undecidable. However, the typing judgements remain similar to Source , meaning that the following three judgements should all be valid and derivable:

$$\begin{aligned} \diamond \vdash \text{AddOne} & : \text{GMaybe}^T \text{Int} \rightarrow \text{GMaybe}^T \text{Int} \\ \diamond \vdash \text{AddOne} & : \text{GMaybe}^F \text{Int} \rightarrow \text{GMaybe}^F \text{Int} \\ \diamond \vdash \text{AddOne} & : \text{GMaybe}^U \text{Int} \rightarrow \text{GMaybe}^U \text{Int} \end{aligned}$$

There are two main implications of this decision:

- typing is not unique in Source° ;
- there are two different constructs: *types* A and *syntactic types* S , where the former have no holes and are used in typing judgements (as before), and the latter have all grades replaced by holes and are used in all annotations (like in function abstractions).

The syntax and typing judgements of Source° are outlined in Figure 4.1. In the syntax, binding and function abstraction terms make use of syntactic types, whereas typing judgements are still of the form $\Gamma \vdash M : A$, meaning that both the typing environment and the judgement itself refer to types. The only typing rules shown are those for the core of Source , and those specific to some graded monad follow the same idea. Syntactic types and types can be related with the *hide* function: a syntactic type S *hides* the grades of type A by replacing all grades with $\circ$. This function is defined recursively on the structure of types:

$$\begin{aligned} \text{hide}(\text{Int}) & = \text{Int} \\ \text{hide}(\text{Unit}) & = \text{Unit} \\ \text{hide}(A \rightarrow B) & = \text{hide}(A) \rightarrow \text{hide}(B) \\ \text{hide}(T^e A) & = T^\circ(\text{hide}(A)) \end{aligned}$$

A derivation of the typing judgement $\diamond \vdash \text{AddOne} : \text{GMaybe}^T \text{Int} \rightarrow \text{GMaybe}^T \text{Int}$ is shown in Figure 4.2. Furthermore, we use a function *addHoles* which replaces grades in terms with holes, defined recursively as follows:

$$\begin{aligned} \text{addHoles}(c) & = c \\ \text{addHoles}(M \text{ op } N) & = \text{addHoles } M \text{ op } N \\ \text{addHoles}(x) & = x \\ \text{addHoles}(\text{fn } x : A \Rightarrow M) & = \text{fn } x : \text{hide}(A) \Rightarrow \text{addHoles } M \\ \text{addHoles}(M \ N) & = (\text{addHoles}(M)) (\text{addHoles}(N)) \end{aligned}$$

Syntax:

(terms)	M, N	$::=$	$n \mid \langle \rangle \mid M \text{ op } N \mid x \mid \text{fn } x : S \Rightarrow M \mid M N$ $\mid \text{return } M \mid \text{bind } x : S \Leftarrow M \text{ with } N \mid \text{sub } M$
(syntactic types)	S	$::=$	$\text{Int} \mid \text{Unit} \mid S_1 \rightarrow S_2 \mid T^\odot S$
(types)	A, B	$::=$	$\text{Int} \mid \text{Unit} \mid A \rightarrow B \mid T^e A$
(grades)	F, G, H	$::=$	$e \mid \mathbf{1} \mid F \odot G$ $\mid o(F_1, \dots, F_k) \quad (\text{for } o \in \mathcal{O} \text{ of arity } k)$

Typing judgements:

(INT)	$\frac{}{\Gamma \vdash n : \text{Int}}$	(UNIT)	$\frac{}{\Gamma \vdash \langle \rangle : \text{Unit}}$
(OP)	$\frac{\Gamma \vdash M : \text{Int} \quad \Gamma \vdash N : \text{Int}}{\Gamma \vdash M \text{ op } N : \text{typefor}(\text{op})}$	(VAR)	$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$
(FN)	$\frac{\Gamma, x : A \vdash M : B \quad S = \text{hide}(A)}{\Gamma \vdash \text{fn } x : S \Rightarrow M : A \rightarrow B}$	(APP)	$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$
(RET)	$\frac{\Gamma \vdash M : A \quad \vdash_{\text{gr}} \mathbf{1} = i}{\Gamma \vdash \text{return } M : T^i A}$	(SUB)	$\frac{\Gamma \vdash M : T^e A \quad \vdash_{\text{gr}} e \leq f}{\Gamma \vdash \text{sub } M : T^f A}$
(BIND)	$\frac{\vdash e \odot f = g \quad \Gamma \vdash M : T^e A \quad \Gamma, x : A \vdash N : T^f B \quad S = \text{hide}(A)}{\Gamma \vdash \text{bind } x : S \Leftarrow M \text{ with } N : T^g B}$		

Figure 4.1. Syntax and the type system of Source° , assuming an underlying analysis-graded monad with functor (grade-indexed type constructor) T and grading algebra $(E, \leq, \odot, \mathbf{1}, \mathcal{O})$.

$$\begin{array}{c}
 \text{(★)} = \frac{\text{(VAR)} \frac{}{x : \text{Int}, m : \text{GMaybe}^F \text{Int} \vdash x : \text{Int}} \quad \text{(INT)} \frac{}{x : \text{Int}, m : \text{GMaybe}^F \text{Int} \vdash \mathbf{1} : \text{Int}} \quad \vdash_{\text{gr}} \mathbf{1} = T}{\text{(RET)} \frac{x : \text{Int}, m : \text{GMaybe}^F \text{Int} \vdash x + 1 : \text{Int}}{x : \text{Int}, m : \text{GMaybe}^F \text{Int} \vdash \text{return } (x + 1) : \text{GMaybe}^T \text{Int}}} \\
 \text{(FN)} \frac{\text{(BIND)} \frac{\vdash F \odot T = F \quad \text{(★)} \quad \text{GMaybe}^\circ \text{Int} = \text{hide}(\text{GMaybe}^F \text{Int}) \quad \text{GMaybe}^\circ \text{Int} \rightarrow \text{GMaybe}^\circ \text{Int} = \text{hide}(\text{GMaybe}^F \text{Int} \rightarrow \text{GMaybe}^F \text{Int})}{m : \text{GMaybe}^F \text{Int} \vdash \text{bind } x : \text{Int} \Leftarrow \text{return } (x + 1) \text{ with } : \text{GMaybe}^F \text{Int}}}{\diamond \vdash \text{AddOne} : \text{GMaybe}^F \text{Int} \rightarrow \text{GMaybe}^F \text{Int}}
 \end{array}$$

Figure 4.2. A typing derivation for term AddOne .

$$\begin{aligned}
\text{addHoles}(\text{return } M) &= \text{return } (\text{addHoles}(M)) \\
\text{addHoles}(\text{sub } M) &= \text{sub}_{\odot, \odot} (\text{addHoles}(M)) \\
\text{addHoles}(\text{bind } x : A \Leftarrow M \text{ with } N) &= \text{bind } x : \text{hide}(A) \Leftarrow \text{addHoles}(M) \text{ with } \\
&\quad \text{addHoles}(N) \\
\text{addHoles}(T^e A) &= T^{\odot}(\text{hide}(A))
\end{aligned}$$

There is a key difference between this approach and making terms *grade-polymorphic*. A $\text{Source}^{\odot}$ typing derivation is essentially the same as a Source typing derivation in which all grades in terms are replaced with $\odot$. Consider the term:

$$\begin{aligned}
&((\text{fn } h : (\text{GMaybe}^{\odot} \text{Int} \rightarrow \text{GMaybe}^{\odot} \text{Int}) \\
&\Rightarrow \text{bind } x \Leftarrow (h \text{ nothing}) \text{ with } h (\text{return } (x + 1))) \text{ AddOne}
\end{aligned}$$

Here h is first applied to `nothing`, whose type is $\text{GMaybe}^{\text{F}} \text{Int}$, and then to $\text{return } (x + 1)$, whose type is $\text{GMaybe}^{\text{T}} \text{Int}$. This is not permitted in the basic $\text{Source}^{\odot}$ type system. This property is common in programming languages with type polymorphism – for example, the following simple OCaml and Haskell expressions do not type-check:

```
(fn h => (h 42, h "foo")) (fn x => x)
```

```
(\h -> (h 42, h "foo")) (\x -> x)
```

This issue is partially resolved by introducing let-expressions and enabling a form of *let-polymorphism*. With polymorphic types the approach is that of introducing *type schemes* in the style of Hindley [Hin69], Damas and Milner [DM82]. We consider polymorphism later in this chapter.

The necessity of subeffecting The $\text{sub } M$ construct exists precisely because of the lack of polymorphism and the fact that $\text{Source}^{\odot}$, like Source , does not define an implicit subtyping rule for terms: if in environment Γ the term M has type A which is *subsumed* by type B , then $\Gamma \vdash M : B$ is also derivable. This is a common idea, but is known to complicate type inference. The seminal overview of the issue is given in Pottier’s PhD thesis. [Pot98] We opt to use sub whenever a term of a graded monadic type *might* need to be upcast or widened – without specifying the grade. The above untypeable example involving `AddOne` is resolved by writing:

$$\begin{aligned}
&((\text{fn } h : (\text{GMaybe}^{\odot} \text{Int} \rightarrow \text{GMaybe}^{\odot} \text{Int}) \\
&\Rightarrow \text{bind } x \Leftarrow (h (\text{sub } \text{nothing})) \text{ with } h (\text{sub } (\text{return } (x + 1)))) \text{ AddOne}
\end{aligned}$$

This approach comes at the cost of imprecision: the only valid typing of this term is $\text{GMaybe}^{\text{U}} \text{Int}$, whereas a more precise answer is that the computation always fails and the grade should be F . Achieving a more precise typing requires polymorphism. In the meantime, we *do* introduce a ordering between types, but only as a tool for talking about type inference.

4.1.2 Analysis-graded monads with $\perp$

Given that terms in context in $\text{Source}^{\odot}$ do not have unique types, the question of inferring grades needs to be properly specified. This is a variant of the usual type inference problem: rather than finding a type with which to substitute ‘?’ in some judgements $\Gamma \vdash M : ?$, the goal is to fill all the missing grade holes $\odot$ in a syntactic type S to produce a type A such that $\Gamma \vdash M : A$ and $A = \text{hide}(S)$. However, if there is more than one solution to a type inference problem, which solution should

we choose? Consider in isolation the function `IdFun` defined as follows in `Source`[®], similar to `AddOne` from before:

$$\text{IdFun} = \text{fn } x : \text{GMaybe}^{\odot} \text{Int} \Rightarrow x$$

Any of the three grades from the carrier set $\{\mathbf{T}, \mathbf{F}, \mathbf{U}\}$ of the grading algebra associated with the `GMaybe` graded monad can be put in place of $\odot$ – so all of the following typing judgements hold:

$$\begin{aligned} \diamond \vdash \text{IdFun} &: \text{GMaybe}^{\mathbf{T}} \text{Int} \rightarrow \text{GMaybe}^{\mathbf{T}} \text{Int} \\ \diamond \vdash \text{IdFun} &: \text{GMaybe}^{\mathbf{F}} \text{Int} \rightarrow \text{GMaybe}^{\mathbf{F}} \text{Int} \\ \diamond \vdash \text{IdFun} &: \text{GMaybe}^{\mathbf{U}} \text{Int} \rightarrow \text{GMaybe}^{\mathbf{U}} \text{Int} \end{aligned}$$

With the framework developed thus far, there is no ‘least’ grade we can use. We address this by representing ‘unexplored’ analysis outcomes (like with abstract interpretation) using a bottom element of the join-semilattice. This is crucial for defining the grade inference framework based on solving lattice equations (Section 4.1.3)

Definition 4.1. For a computational grading algebra $\mathcal{E} = (E, \leq, \odot, \mathbf{1}, \mathcal{O})$, there is a $\perp$ -augmented computational grading algebra $\mathcal{E}_{\perp} = (E_{\perp}, \leq_{\perp}, \odot_{\perp}, \mathbf{1}, \mathcal{O}_{\perp})$ with $E_{\perp} = E \cup \{\perp\}$ for $\perp \notin E$, $\perp \leq_{\perp} e$ for all $e \in E$, $\perp \odot e = \perp = e \odot \perp$ for $e \in E$, and similarly extended operators in $\mathcal{O}$.

For convenience, we refer to the relation $\leq_{\perp}$, operator $\odot_{\perp}$ and set $\mathcal{O}_{\perp}$ as simply $\leq$, $\odot$ and $\mathcal{O}$. For the remainder of this chapter, we consider variants of graded monads, but graded with $\perp$ -augmented computational grading algebras.

Translating graded monadic types $T^{\perp}A$ from `Source` to `Target` requires a way of achieving explicit coercion of terms from $T^{\perp}A$ to T^eA . As grade $\perp$ represents ‘not explored’ and not match any actual computation, the full translation of type $T^{\perp}A$ in `Source` should be to an *uninhabited type*. Thus we add type `Emp` to `Target`, which is uninhabited – there is no constant of that type. The translation is then as follows:

$$\langle\langle T^{\perp}A \rangle\rangle_{\text{type}} = \text{Emp}$$

Categorically, `Emp` corresponds to the initial object in the category of discourse.

4.1.3 The grade inference framework

Given that types are not unique in `Source`[®], the question of inferring grades needs to be properly specified. This is a variant of the usual type inference problem: rather than finding a type with which to substitute ‘?’ in some judgements $\Gamma \vdash M : ?$, the goal is to fill all the missing grade holes $\odot$ in a syntactic type S to produce a type A such that $\Gamma \vdash M : A$ and $A = \text{hide}(S)$. Out of all valid typing judgements that hold for a term M in a typing environment Γ , we are interested in finding a type A such that $\Gamma \vdash M : A$ such that, given any other type B for which $\Gamma \vdash M : B$ is derivable, the grade-erased types A and B are the same, and all the grades of A are no greater than the corresponding grades in B . We call this solution A the *principal* type (which exists by Theorem 4.1), and we correspondingly define a *subgrading* relation $<|$ between types. The solution A is the least solution of the typing problem. The definition of this relation is given in Figure 4.3.

$$\begin{array}{c}
\text{(REFL)} \frac{}{A <| A} \qquad \text{(TRANS)} \frac{A <| B \quad B <| C}{A <| C} \\
\text{(FUN)} \frac{A <| A' \quad B <| B'}{A \rightarrow B <| A' \rightarrow B'} \qquad \text{(GM)} \frac{\vdash_{\text{sub}} e \leq f \quad A <| B}{T^e A <| T^f B}
\end{array}$$

Figure 4.3. The definition of the subgrading relation on types.

In this setup, a solution to the grade inference problem is the *least* one, meaning that for a context Γ and term M it is a type A such that $\Gamma \vdash M : A$ and for any other type B satisfying $\Gamma \vdash M : B$ we have $A <| B$.

The subgrading relation is *not* the same as type subsumption seen in type systems with subeffecting. This is best seen from the fact that subgrading for function types is covariant both in the argument and result, whereas subsumption is contravariant in the argument. A function whose type is $\text{PermState}_{\text{Int}}^{\text{ro}} \text{Int} \rightarrow \text{PermState}_{\text{Int}}^{\text{wo}} \text{Int}$ cannot be applied directly to `return 1` (as its type is $\text{PermState}_{\text{Int}}^{\text{pure}} \text{Int}$ in this case), but the programmer can ‘overapproximate’ the argument and write `sub (return 1)` instead. Subgrading emphasises the idea of conservative overapproximation, whereas subtyping places a minimum requirement on the argument.

We now turn to a method of finding a solution to the grade inference problem for `Source`[®] as presented, given an underlying analysis-graded monad T with a $\perp$ -augmented computational grading algebra $\mathcal{E}_{\perp}$. The method is *constraint-based*: all grades in the holes of syntactic types and are replaced with fresh *grade variables* (ranging over $r_1, \dots, r_k, \dots$), which are then instantiated with concrete grades by solving a set of equality and inequality constraints. Generating those constraints is done using three functions: $\text{typeFor}(\Gamma \vdash M : ?)$, $\text{unhide}(S)$ and $\text{eqTypes}(A, B)$. where the first returns a pair of type and set of constraints, the second turns a syntactic type into a type with fresh grade variable names, and $\text{eqTypes}(A, B)$ gives the (unique) set of constraints that make types A and B equal. The definitions of these functions are given in Figure 4.4. In particular we see that the definition of $\text{typeFor}(\Gamma \vdash M : ?)$ follows the structure of typing derivations, so for any additional constructs the same principle applies: all of the side conditions based on $\vdash$ and $\vdash_{\text{sub}}$ turn into constraints.

Theorem 4.1. Given a term M with no free variables, if $\diamond \vdash M : A'$ in `Source` for some type A' , then $\text{typeFor}(\diamond \vdash \text{addHoles}(M) : ?) = (A, C)$ for some A and C , $\text{hide}(A) = \text{hide}(A')$, and there is an instantiation of all grade variables of A giving A' and which satisfies all the constraints in C .

Proof. We prove a slightly stronger proposition Φ ranging over typing judgements:

For $\Gamma \vdash M : A'$ in `Source`, there exist A and C such that $\text{typeFor}(\Gamma \vdash \text{addHoles}(M) : ?) = (A, C)$, $\text{hide}(A) = \text{hide}(A')$, and there is an instantiation of all grade variables of A giving A' satisfying all the constraints in C .

We prove this statement by induction over typing derivations in `Source`. We only state two of the interesting cases:

$typeFor(\Gamma \vdash \langle \rangle : ?)$	$= (\text{Unit}, \emptyset)$
$typeFor(\Gamma \vdash n : ?)$	$= (\text{Int}, \emptyset)$
$typeFor(\Gamma \vdash x : ?)$	$= (\Gamma(x), \emptyset)$
$typeFor(\Gamma \vdash \text{fn } x : S \Rightarrow M : ?)$	$= \text{let } A = \text{unhide}(S) \text{ in}$ $\text{let } (B, C) = typeFor(\Gamma, x : A \vdash M : ?) \text{ in}$ $(A \rightarrow B, C)$
$typeFor(\Gamma \vdash M N : ?)$	$= \text{let } (A_1, C_1) = typeFor(\Gamma \vdash M : ?) \text{ in}$ $\text{let } (A_2, C_2) = typeFor(\Gamma \vdash N : ?) \text{ in}$ $\text{let } (B_1 \rightarrow B_2) = A_1 \text{ in}$ $\text{let } C_3 = eqTypes(B_1, A_2) \text{ in}$ $(B_2, C_1 \cup C_2 \cup C_3)$
$typeFor(\Gamma \vdash \text{return } M : ?)$	$= \text{let } (A, C) = typeFor(\Gamma \vdash M : ?) \text{ in}$ $\text{let } r = \text{fresh} \text{ in}$ $(T^r A, C \cup \{r = \mathbf{1}\})$
$typeFor(\Gamma \vdash \text{sub } M : ?)$	$= \text{let } (T^{r_1} A, C) = typeFor(\Gamma \vdash M : ?) \text{ in}$ $\text{let } r_2 = \text{fresh} \text{ in}$ $(T^{r_2} A, C \cup \{r_1 \leq r_2\})$
$typeFor(\Gamma \vdash \text{bind } x : S \Leftarrow M \text{ with } N)$	$= \text{let } r = \text{fresh} \text{ in}$ $\text{let } (T^{r_1} A_1, C_1) = typeFor(\Gamma \vdash M : ?) \text{ in}$ $\text{let } (T^{r_2} A_2, C_2) =$ $\quad typeFor(\Gamma, x : A_1 \vdash N : ?) \text{ in}$ $\text{let } C_3 = eqTypes(A_1, \text{unhide}(S)) \text{ in}$ $(T^r A_2, C_1 \cup C_2 \cup C_3 \cup \{r = r_1 \odot r_2\})$
$unhide(\text{Unit})$	$= \text{Unit}$
$unhide(\text{Int})$	$= \text{Int}$
$unhide(S_1 \rightarrow S_2)$	$= \text{unhide}(S_1) \rightarrow \text{unhide}(S_2)$
$unhide(T^{\odot} S)$	$= \text{let } r = \text{fresh} \text{ in } T^r(\text{unhide}(S))$
$eqTypes(\text{Unit}, \text{Unit})$	$= \emptyset$
$eqTypes(\text{Int}, \text{Int})$	$= \emptyset$
$eqTypes(A_1 \rightarrow A_2, B_1 \rightarrow B_2)$	$= eqTypes(A_1, B_1) \cup eqTypes(A_2, B_2)$
$eqTypes(T^{r_1} A_1, T^{r_2} A_2)$	$= eqTypes(A_1, A_2) \cup \{r_1 = r_2\}$

Figure 4.4. The algorithms for $typeFor(\cdot)$, $unhide(\cdot)$ and $eqTypes(\cdot)$. If at any point a call fails (for example, $eqTypes()$ is given incompatible arguments), the algorithm fails.

(fn) We make use of a lemma in this case:

Lemma 4.1. If $B = \text{unhide}(\text{hide}(A))$, then there exists an assignment ψ such that $\psi B = A$.

Proof. Constructively, by induction on the structure of types A . Each fresh variable generated with $\text{unhide}()$ is mapped to the appropriate grade in A . $\square$

Assume $\Gamma, x : D_1 \vdash M : D_2$ and $\Phi(\Gamma, x : D_1 \vdash M : D_2)$.

Consider the term $\text{fn } x : S \Rightarrow M$, where $S = \text{hide}(D_1)$. Let $A = \text{unhide}(S)$ and let $(B, C) = \text{typeFor}(\Gamma, x : A \vdash \text{addHoles}(M) : ?)$. By the above Lemma, since $A = \text{unhide}(S) = \text{unhide}(\text{hide}(D_1))$, there is an assignment ψ such that $\psi A = D_1$. Then $\text{typeFor}(\Gamma, x : D_1 \vdash \text{addHoles}(M) : ?) = (\psi B, \psi C)$.

By the induction hypothesis, there is an assignment φ such that $\varphi \psi B = D_2$ which is consistent with ψC and such that $\Gamma, x : D_1 \vdash M : D_2$. Hence, $\varphi \psi(A \rightarrow B) = D_1 \rightarrow D_2$.

By (FN), we have $\Gamma \vdash \text{fn } x : S \Rightarrow M : D_1 \rightarrow D_2$. Since $\varphi \psi(A \rightarrow B) = D_1 \rightarrow D_2$ and $\varphi \psi$ is consistent with C , we conclude $\Phi(\Gamma \vdash M : D_1 \rightarrow D_2)$ holds.

(app) Assume that judgements $\Gamma \vdash M : D'_1 \rightarrow D'_2$ and $\Gamma \vdash N : D'_1$ hold, and that $\Phi(\Gamma \vdash M : D'_1 \rightarrow D'_2)$ and $\Phi(\Gamma \vdash N : D'_1)$.

By (APP) we have $\Gamma \vdash M N : D'_2$.

Let $(A_1, C_1) = \text{typeFor}(\Gamma \vdash \text{addHoles}(M) : ?)$ and $(A_2, C_2) = \text{typeFor}(\Gamma \vdash \text{addHoles}(N) : ?)$.

By assumption, $\text{hide}(A_1) = \text{hide}(D'_1 \rightarrow D'_2) = \text{hide}(D'_1) \rightarrow \text{hide}(D'_2)$, and $\text{hide}(A_2) = \text{hide}(D'_1)$, and there are assignments φ_1, φ_2 to grade variables such that $D'_1 \rightarrow D'_2 = \varphi_1 A_1$ and $D'_1 = \varphi_2 A_2$.

By the rules defining hide , the type A_1 is necessarily of the form $B_1 \rightarrow B_2$. Then $\text{hide}(A_1) = \text{hide}(B_1) \rightarrow \text{hide}(B_2)$. It follows that $\text{hide}(B_1) = \text{hide}(D'_1)$, $\text{hide}(B_2) = \text{hide}(D'_2)$, and also $\varphi_1(B_1 \rightarrow B_2) = D'_1 \rightarrow D'_2$. Thus, $\varphi_1 B_2 = D'_2$.

It remains to show that φ_1 is consistent with $C_1 \cup C_2 \cup C_3$. φ_1 is consistent with C_1 by the induction hypothesis. φ_1 is consistent with C_2 because $\varphi_2 A_2 = \varphi_1 B_1 = D'_1$. Finally, the constraints generated by C_3 are satisfied by φ_1 because $\varphi_1 B_1 = D'_1 = \varphi_2 A_2$. Thus, $\Phi(\Gamma \vdash M N : D'_2)$.

$\square$

Example 4.1. Consider the term:

$$\text{Ex} = \text{fn } m : \text{PermState}_{\text{Int}}^{\circ} \text{Int} \Rightarrow \text{bind } x \leftarrow m \text{ with put } (x + 1)$$

In an empty context, the following are valid types for Ex :

$$\begin{array}{ll}
 \text{PermState}_{\text{Int}}^{\perp} \text{Int} & \rightarrow \text{PermState}_{\text{Int}}^{\text{aw}} \text{Int} \\
 \text{PermState}_{\text{Int}}^{\text{pure}} \text{Int} & \rightarrow \text{PermState}_{\text{Int}}^{\text{aw}} \text{Int} \\
 \text{PermState}_{\text{Int}}^{\text{ro}} \text{Int} & \rightarrow \text{PermState}_{\text{Int}}^{\text{rw}} \text{Int} \\
 \text{PermState}_{\text{Int}}^{\text{aw}} \text{Int} & \rightarrow \text{PermState}_{\text{Int}}^{\text{aw}} \text{Int} \\
 \text{PermState}_{\text{Int}}^{\text{wo}} \text{Int} & \rightarrow \text{PermState}_{\text{Int}}^{\text{aw}} \text{Int} \\
 \text{PermState}_{\text{Int}}^{\text{rw}} \text{Int} & \rightarrow \text{PermState}_{\text{Int}}^{\text{rw}} \text{Int}
 \end{array}$$

Meanwhile, $\text{typeFor}(\diamond \vdash \text{Ex} : ?)$ returns the following tuple:

$$(\text{PermState}_{\text{Int}}^{r_1} \text{Int} \rightarrow \text{PermState}_{\text{Int}}^{r_2} \text{Int}, \{r_2 = r_1 \odot r_3, r_3 = \text{aw}\})$$

All valid types are obtained by different instantiations of the grade variables (corresponding to all six possible values of r_1).

4.1.4 Solving constraints

So far we have only presented which constraints get generated and how. What is missing is a method of finding a solution. For a given system of simultaneous equations and inequations in $\perp$ -augmented computational grading algebra, we can take two approaches:

1. The theory associated with the $\perp$ -augmented computational grading algebra has a decidable theory of finding least solutions to systems of equations and inequations. For example, HistState_S has as its grading algebra $(\mathbb{N}_\omega, \leq_\omega, +_\omega, 0, \{1\})$ and an equation such as $r = r +_\omega 1$ is known to have least solution $r = \omega$, whereas any $e \in \mathbb{N}_\omega$ satisfies $r \leq_\omega r + 1$.
2. Use Tarski-style iteration to compute the least solution, whenever such iteration converges to the least solution.

The former approach assumes an underlying decision procedure, which subsumes the second method (since it is also a decision procedure). We do not focus on it and instead turn immediately to the iteration approach, as it is classical in program analysis.

Firstly, note that the generated constraints are either the form $r = F$ or $r \geq G$ for some grade variable r and grade expression F . A single grade variable can thus have multiple such constraints: $r = F_1, r = F_2, \dots, r = F_m, r \geq G_1, \dots, r \geq G_n$. These are merged into a single constraint $r = F_1 \vee \dots \vee F_m \vee G_1 \vee \dots \vee G_n \vee r$, as the right-hand side is the least grade satisfying all the constraints. The last ' $\vee r$ ' is added in order to account for unconstrained grade variables. Seeing how the grade expressions depend on all grade variables introduced, say $r_1, \dots, r_k$, and that we can repeat this constraint merging for every grade variable, we end up with the following set of constraints, for some functions $u_1, \dots, u_k$:

$$r_1 = u_1(r_1, \dots, r_k), \quad \dots, \quad r_k = u_k(r_1, \dots, r_k)$$

We say the derived set of constraints – one equality constraint per grade variable, constraint merging via joins – gives the *join-form* of the original set of constraints C .

Lemma 4.2. Any solution to the original set of constraints C is also a solution of the join-form of C .

More concisely, we can use vector notation and say $\vec{r} = (r_1, \dots, r_k)$, and introduce a vector function u such that $u(\vec{r}) = (u_1(\vec{r}), \dots, u_k(\vec{r}))$. The individual u_i functions are all monotonic because they consist of joins and grade expressions are monotonic functions in their grade variables. This means u is also a monotonic function in the product domain: for $\vec{r} \leq \vec{r}'$ (meaning $r_1 \leq r'_1, \dots, r_k \leq r'_k$), $u(\vec{r}) \leq u(\vec{r}')$.

Based on the fact that u is monotonic and Theorem 2.2, we have the following theorem stating that an approach similar to Knaster-Tarski iteration gives a solution to the constraint set:

Theorem 4.2. Assuming a finite-height $\perp$ -augmented computational grading algebra, the sequence sequence $\vec{r}^{(i)}$ defined with $\vec{r}^{(0)} = (\perp, \dots, \perp)$ and $\vec{r}^{(i+1)} = u(\vec{r}^{(i)})$ eventually converges.

The key condition here is *finite-height*, as it guarantees the iterative procedure converges.

This gives a simple algorithm for solving constraints:

1. convert all equality and inequality constraint into the join-form;
2. initialise all grade variables in $\vec{r}$ to $\perp$;
3. while $\vec{r} \neq u(\vec{r})$, assign $\vec{r} \leftarrow u(\vec{r})$;
4. check that the individual equality and inequality constraints are satisfied: if they are, return successfully; otherwise, return ‘no solution’.

The last step is necessary to avoid issues stemming from an inconsistent set of equality constraints such as $\{r = \mathbf{T}, r = \mathbf{F}\}$: the iteration is $u(r) = \mathbf{T} \vee \mathbf{F} \vee r$, whose least fixed-point is $\mathbf{U}$, yet it is inconsistent with the constraints.

Example 4.2. Consider the following example term:

```
fn x : Int ⇒ alt x ≥ 0 then (return x) else (bind _ : Unit ⇐ put x with return 1)
```

Suppose r_0 is the grade associated with the whole conditional expression with **alt**. Let the grade variable associated with the **then** branch be r_1 , the grade variable associated with the **else** branch r_2 , and let r_3 be the grade variable associated with **put x** and r_4 the grade associated with **return 1**. The following constraints get generated:

$$\begin{aligned} r_0 &= r_1 \vee r_2, & r_1 &= \text{pure}, & r_2 &= r_3 \odot r_4, \\ r_3 &= \text{aw}, & r_4 &= \text{pure}. \end{aligned}$$

and the type is $\text{Int} \rightarrow \text{PermState}_{\text{Int}}^{r_0} \text{Int}$. The iteration function u is thus:

$$u(r_0, r_1, r_2, r_3, r_4) = (r_1 \vee r_2 \vee r_0, \text{pure} \vee r_1, (r_3 \odot r_4) \vee r_2, \text{aw} \vee r_3, \text{pure} \vee r_4)$$

Starting with $\vec{r}^{(0)} = (\perp, \perp, \perp, \perp, \perp)$, we have:

$$\begin{aligned}\vec{r}^{(1)} &= (\perp, \text{pure}, \perp, \text{aw}, \text{pure}) \\ \vec{r}^{(2)} &= (\text{pure}, \text{pure}, \text{aw}, \text{aw}, \text{pure}) \\ \vec{r}^{(3)} &= (\text{wo}, \text{pure}, \text{aw}, \text{aw}, \text{pure}) \\ \vec{r}^{(4)} &= (\text{wo}, \text{pure}, \text{aw}, \text{aw}, \text{pure})\end{aligned}$$

These grades are consistent with the original constraints. Thus the inferred type is $\text{Int} \rightarrow \text{PermState}_{\text{Int}}^{\text{wo}} \text{Int}$.

4.1.5 Adding recursion

There are various ways in which programming languages and calculi syntactically represent recursion. The approach we opt for is the one that allows us to write both recursive functions and recursive graded monadic computations, using *least fixed-point* syntax.

We add terms of the form $\text{fix } M$ to Target, Source and Source[®] to represent such recursive computations. In both of these languages, the typing judgement associated with the construct is the following:

$$(\text{FIX}) \frac{\Gamma \vdash M : A \rightarrow A}{\Gamma \vdash \text{fix } M : A}$$

and the transition semantics in Target is given as follows:

$$\text{fix } (\text{fn } x : A \Rightarrow M) \rightsquigarrow M\{(\text{fix } (\text{fn } x : A \Rightarrow M))/x\}$$

The only remaining part is the Source-to-Target translation, which is very simple due to the overlap in syntax:

$$\langle\langle \Gamma \vdash \text{fix } M : A \rangle\rangle_{\text{val}} = \text{fix } \langle\langle \Gamma \vdash M : A \rightarrow A \rangle\rangle_{\text{val}}$$

Fundamentally, introducing recursion on terms does not affect the type inference discussion from before, when grades could depend on themselves – there are already possibly cyclic dependencies. The only things that needs to be introduced is the definition of $\text{typeFor}(\cdot)$:

$$\begin{aligned}\text{typeFor}(\Gamma \vdash \text{fix } M : ?) &= \text{let } (A \rightarrow B, C) = \text{typeFor}(\Gamma \vdash M : ?) \text{ in} \\ &\quad (B, C \cup \text{eqTypes}(A, B))\end{aligned}$$

Example 4.3. Consider the following two Source[®] terms:

$$\begin{aligned}\text{RecOnce} &= \text{fix } (\text{fn } h : (\text{Int} \rightarrow \text{Int} \rightarrow \text{HistState}_{\text{Int}}^{\text{®}} \text{Unit}) \Rightarrow \\ &\quad \text{fn } x : \text{Int} \Rightarrow \text{fn } s : \text{Int} \Rightarrow \\ &\quad \text{if } x \geq 0 \text{ then put } s \\ &\quad \quad \text{else } (\text{bind } y \leftarrow \text{get with } h \ (x - 1) \ (s + y))) \\ \text{RecOmega} &= \text{bind } y : \text{Int} \leftarrow \text{get with} \\ &\quad \text{fix } (\text{fn } h : (\text{Int} \rightarrow \text{Int} \rightarrow \text{HistState}_{\text{Int}}^{\text{®}} \text{Unit}) \Rightarrow \\ &\quad \text{fn } x : \text{Int} \Rightarrow \text{fn } s : \text{Int} \Rightarrow \\ &\quad \text{if } x \geq 0 \text{ then return } \langle \rangle \\ &\quad \quad \text{else } (\text{bind } t \leftarrow \text{put } (s + y) \text{ with } h \ (x - 1) \ (s + y)))\end{aligned}$$

Semantically, these terms represent two very similar recursive functions of two arguments x and s . The former performs exactly one write to the store, whereas the latter performs updates in every recursive call and is bounded only by the value of the passed argument. This means that the associated grades are 1 and ω , as seen in the typing judgements that can be derived:

$$\begin{aligned} \Gamma \vdash \text{RecOnce} & : \text{Int} \rightarrow \text{Int} \rightarrow \text{HistState}_{\text{Int}}^1 \text{Unit} \\ \Gamma \vdash \text{RecOmega} & : \text{Int} \rightarrow \text{Int} \rightarrow \text{HistState}_{\text{Int}}^{\omega} \text{Unit} \end{aligned}$$

4.2 Polymorphic grades

Thus far the approach to grade inference was based on the idea of finding a least solution, which corresponds to the idea of finding the most precise analysis answer. This was done in a setting without *grade-parametric polymorphism*, which in practice means that a function expecting a graded monadic computation as an argument only accepts those whose type is $T^e A$ for a particular grade e , as opposed to several compatible grades. The first such example seen in this chapter is that of `AddOne`, described in section 4.1.1. This problem is already known and explored in the setting of both type systems with polymorphism and in effect systems, with the solution being the introduction of type variables [DM82] and effect variables [LG88]. We adapt this approach to grades.

We first define yet another variant of `Source`, which we call `PolySource`, which syntactically resembles `Source⊙`, but with a fundamentally different type system and with the addition of a `let` keyword similar to that in ML-like languages and Haskell, providing let-polymorphism in the style of Hindley, Milner and Damas. We then define a grade inference algorithm based on unification – similar to the algorithms used in polymorphic type systems.

4.2.1 Grade-polymorphic programming

The key idea of `PolySource` is that the grade variables, as seen in the type inference algorithm for `Source⊙`, become an integral part of the type system. This means that the way grades are treated in the language change in two significant ways: firstly, graded monadic types can now use any grade terms instead of just specific algebra elements; secondly, types are polymorphic in grade variables, which are bound in $\forall_{\text{eff}}$ constructs.

We define the notion of *type schemes* σ as constructs of the form $\forall_{\text{eff}} R(A)$, where R is a set of names of grade variables (ranging over r, s) which can appear in A – for example, $\forall_{\text{eff}} r_1, r_2 (\text{PermState}_{\text{Int}}^{r_1 \odot (1 \vee r_2)} \text{Unit})$ is a type scheme in `PolySource`. Similarly, the type scheme corresponding to the `AddOne` example is $\forall_{\text{eff}} r (\text{GMaybe}^r \text{Int} \rightarrow \text{GMaybe}^r \text{Int})$. With the addition of type schemes, the typing environment Γ changes: it is no longer a mapping from variable names to types, but to type schemes. As type schemes in `PolySource` represent a range of possible types, membership to the set of types given by the scheme is described using the *specialisation* relation $\succ$.

Definition 4.2. A *grade substitution* θ is a substitution of some grade variables to grade terms.

Definition 4.3. We say type A specialises a type scheme σ if $\sigma = \forall_{\text{eff}} R(B)$ and there is a grade substitution θ whose domain is R such that $A = B\theta$.

We adopt notation $\Gamma \vdash M : \forall_{\text{eff}} R(A)$ to mean that $\Gamma \vdash M : A$, where R is the set of grade variables names appearing freely in A that do not appear freely in Γ . The function $\text{fev}(\cdot)$ represents the free grade variables in a type or a context, so $R = \text{fev}(A) \setminus \text{fev}(\Gamma)$.

We furthermore adopt the notation θA to represent the application of θ to A , where all free occurrences of grade variables in A are replaced according to θ . Next, $\theta\sigma$ substitutes free occurrences of grade variables in a type scheme σ , renaming the bound variables if necessary. Furthermore, we write $\theta\Gamma$ to mean ‘ Γ , with θ applied to all type schemes that form the context’. We also write $\theta_2\theta_1$ to represent composition of θ_2 and θ_1 .

The syntax and the typing rules are given in Figure 4.5. They are very similar to Source° , but with several important distinctions. Firstly, the sub is not part of the syntax of PolySource , because it is no longer necessary – in Source° it was used to pass as an argument a term of type $T^e A$ to a function expecting an input whose type is $T^f A$, where $e \leq f$. Polymorphism resolves this issue in a significant number of situations. We also omit it because polymorphism and subtyping make the presentation somewhat more difficult, as we discuss later in this section. Secondly, a polymorphic let -expression construct is introduced into the language, whose typing rule is similar to that seen in the conventional let constructs. Thirdly, there are no longer any side conditions regarding grades – instead, the expressions appearing in the grades are spelled out in the syntax of PolySource (for example, $F \odot G$ in the (BIND) rule). Finally, a *type equality* rule is introduced, labelled (TYPE-EQ) , stating that if types A and B are equal and M is a term of type A given Γ , then M also has type B given Γ .

Let us briefly consider the type equality relation. Type equality allows us to say that, for example, $\text{GMaybe}^{\top \odot \top} \text{Unit}$ and $\text{GMaybe}^{\top} A$ are the exact same type (recall $\top$ is the top element), meaning that grade simplification is possible. The definition of the type equality relation $\equiv_{\text{type}}$ is simple and structural, and is given in Figure 4.6. The only interesting aspect of this relation is in the graded monadic case, where the real ‘hard work’ is done: this relation leverages decidability of equality in the grading algebra, as previously seen in $\vdash_{\text{gr}} F = G$ judgements. In short, this notion of type equality just lifts the grade equality judgements to the type level, giving a congruence property.

Translating to Target

The semantics of PolySource is once again stated in terms of the translation function $\llbracket \Gamma \vdash M : A \rrbracket_{\text{val}}$ giving Target terms, but now the translation is limited to monomorphic types A that are also *fully simplified*, meaning that all grades in A are elements of the grading algebra (as opposed to general grade expressions) and there are no free grade variables in either A or Γ . Other approaches to defining the semantics exist (discussed later), this one keeps the definition minimal and keeps this consistent with the approach of Source . The key changes occur whenever there are significant changes compared to Source .

The definition of the translation is brief, as it has no significant changes compared to the one of Source . The only new construct to translate are the let terms – translated

Syntax:

(terms)	M, N	$::=$	$n \mid \langle \rangle \mid M \text{ op } N \mid x \mid \text{fn } x : S \Rightarrow M \mid M N$ $\mid \text{return } M \mid \text{bind } x : S \Leftarrow M \text{ with } N$ $\mid \text{let } x : S = M \text{ in } N$
(syntactic types)	S	$::=$	$\text{Int} \mid \text{Unit} \mid S_1 \rightarrow S_2 \mid T^{\odot} S$
(types)	A, B	$::=$	$\text{Int} \mid \text{Unit} \mid A \rightarrow B \mid T^F A$
(type schemes)	σ	$::=$	$\forall R(A)$
(grades)	F, G, H	$::=$	$e \mid \mathbf{1} \mid r \mid F \odot G$ $\mid o(F_1, \dots, F_k) \quad (\text{for } o \in \mathcal{O} \text{ of arity } k)$

Typing judgements:

(INT)	$\frac{}{\Gamma \vdash n : \text{Int}}$	(UNIT)	$\frac{}{\Gamma \vdash \langle \rangle : \text{Unit}}$
(OP)	$\frac{\Gamma \vdash M : \text{Int} \quad \Gamma \vdash N : \text{Int}}{\Gamma \vdash M \text{ op } N : \text{typefor}(\text{op})}$	(VAR)	$\frac{x : \sigma \in \Gamma \quad \sigma > A}{\Gamma \vdash x : A}$
(FN)	$\frac{\Gamma, x : A \vdash M : B \quad S = \text{hide}(A)}{\Gamma \vdash \text{fn } x : S \Rightarrow M : A \rightarrow B}$	(APP)	$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B}$
(RET)	$\frac{\Gamma \vdash M : A}{\Gamma \vdash \text{return } M : T^1 A}$	(TYPE-EQ)	$\frac{\Gamma \vdash M : A \quad A \equiv_{\text{type}} B}{\Gamma \vdash M : B}$
(LET)	$\frac{\Gamma \vdash M : A \quad \Gamma, x : \forall R(A) \vdash N : B \quad R = \text{fev}(A) \setminus \text{fev}(\Gamma) \quad S = \text{hide}(A)}{\Gamma \vdash \text{let } x : S = M \text{ in } N : B}$		
(BIND)	$\frac{\Gamma \vdash M : T^F A \quad \Gamma, x : A \vdash N : T^G B \quad S = \text{hide}(A)}{\Gamma \vdash \text{bind } x : S \Leftarrow M \text{ with } N : T^{F \odot G} B}$		

Figure 4.5. Syntax and the type system of PolySource, assuming an underlying analysis-graded monad $\mathbf{T}$ with functor (type constructor) T and grading algebra $(E, \leq, \odot, \mathbf{1}, \mathcal{O})$.

($\equiv_{\text{type-INT}}$)	$\frac{}{\text{Int} \equiv_{\text{type}} \text{Int}}$	($\equiv_{\text{type-BOOL}}$)	$\frac{}{\text{Bool} \equiv_{\text{type}} \text{Bool}}$
($\equiv_{\text{type-FN}}$)	$\frac{A \equiv_{\text{type}} A' \quad B \equiv_{\text{type}} B'}{A \rightarrow B \equiv_{\text{type}} A' \rightarrow B'}$	($\equiv_{\text{type-GM}}$)	$\frac{A \equiv_{\text{type}} A' \quad \vdash_{\text{gr}} F = G}{T^F A \equiv_{\text{type}} T^G A'}$

Figure 4.6. The inductive definition of the type equality relation.

by unwrapping **let** into a substitution:

$$\langle\langle\Gamma \vdash \text{let } x : S = M \text{ in } N : B\rangle\rangle_{\text{val}} = \langle\langle\Gamma \vdash N\{(\langle\langle\Gamma \vdash M : A\rangle\rangle_{\text{val}})/x\} : B\rangle\rangle_{\text{val}}$$

The translation of **let**-expressions is predicated on the idea that a judgement $\Gamma \vdash \text{let } x : S = M \text{ in } N : B$ being derivable, with $\Gamma \vdash M : A$ as a premise, implies that $\Gamma \vdash N\{M/x\} : B$ is also derivable. As the other premise in the derivation is $\Gamma, x : \forall_{\text{eff}} R(A) \vdash N : B$ (with $R = \text{fev}(A) \setminus \text{fev}(\Gamma)$), this is just an immediate consequence of a more general substitution lemma:

Lemma 4.3. For terms M, N , types A, B , environment Γ and type scheme σ such that $\Gamma \vdash M : A$ and $\Gamma, x : \sigma \vdash N : B$ are derivable and $\sigma > A$, we have that $\Gamma \vdash N\{M/x\} : B$ is derivable.

Proof. By induction on typing derivations $\Gamma, x : \sigma \vdash N : B$. □

The most significant difference is the need for monomorphisation. Grade-polymorphic variables in Γ need to be specifically replaced with appropriate monomorphic instances: a type scheme $\forall_{\text{eff}} r(\text{GMaybe}^r \text{Int} \rightarrow \text{GMaybe}^{r \odot \top} \text{Int})$ for a variable x means x can be used in several contexts. Monomorphisation therefore turns one variable x into as many as there are possible valuations of the grade variables – for **GMaybe** that means three: $x_{\top}, x_{\text{F}}, x_{\text{U}}$. In general, this only means variable terms are translated:

$$\langle\langle\Gamma \vdash x : A\rangle\rangle_{\text{val}} = x_{\vec{e}}, \quad \text{for } x : \forall_{\text{eff}} R(B) \in \Gamma \text{ and } A = B\{\vec{e}/R\}$$

but translating context is more sophisticated: a single $x : \sigma$ is replaced by a whole list of possible monomorphic versions of x . Of course, this does not work for infinite grading algebras – which remains an incompleteness in the translation, which for now we accept.

4.2.2 Grade inference

As is the case with language with Hindley-Milner-Damas polymorphic types, Poly-Source has a decidable type inference algorithm based on unification. Even in a polymorphic setting, we are able to present type inference as a way of performing a program analysis.

The algorithm is described in terms of a function *infer*($\cdot$). It is defined on partial typing judgements $\Gamma \vdash M : ?$ and returns either a tuple (θ, A) or **FAIL**. The former result, when returned, should mean $\theta\Gamma \vdash M : A$ is derivable, whereas returning **FAIL** means that the term M is not typeable in context Γ . The definition of this function, and thus the algorithm, is given in Figure 4.7. It is somewhat similar to the definition of *typeFor*($\cdot$), with the key difference being that the most general grades are computed within the context of a term, as opposed to generating constraints which get solved later.

As is the case with polymorphic types, the type scheme determined by the algorithm is the *principal* one, meaning that every other valid typing judgement for M in environment Γ is merely a specialisation of the returned solution.

(Here we use fresh sets (α -rename) for R at each use of x)	
$infer(\Gamma \vdash x : ?)$	$= \text{let } \forall_{\text{eff}} R(A) = \Gamma(x) \text{ in } (Id, A)$
$infer(\Gamma \vdash n : ?)$	$= (Id, \text{Int})$
$infer(\Gamma \vdash \langle \rangle : ?)$	$= (Id, \text{Unit})$
$infer(\Gamma \vdash \text{fn } x : S \Rightarrow M : ?)$	$= \text{let } \alpha = \text{fresh in}$ $\text{let } (\theta, A) = infer(\Gamma, x : \alpha \vdash M : ?) \text{ in}$ $(\theta, \theta(\alpha) \rightarrow A)$
$infer(\Gamma \vdash \text{return } M : ?)$	$= \text{let } (\theta, A) = infer(\Gamma \vdash M : ?) \text{ in}$ $(\theta, M^1 A)$
$infer(\Gamma \vdash M_1 M_2 : ?)$	$= \text{let } \alpha = \text{fresh in}$ $\text{let } (\theta_1, A_1) = infer(\Gamma \vdash M_1 : ?) \text{ in}$ $\text{let } (\theta_2, A_2) = infer(\theta_1 \Gamma \vdash M_2 : ?) \text{ in}$ $\text{let } \theta_3 = unify(\theta_2 A_1, A_2 \rightarrow \alpha) \text{ in}$ $(\theta_3 \theta_2 \theta_1, \theta_3(\alpha))$
$infer(\Gamma \vdash \text{bind } x : S \Leftarrow M_1 \text{ with } M_2 : ?)$	$= \text{let } (\theta_1, A_1) = infer(\Gamma \vdash M_1 : ?) \text{ in}$ $\text{let } (\theta_2, A_2) =$ $infer(\theta_1 \Gamma, x : A_1 \vdash M_2 : ?) \text{ in}$ $\text{let } \alpha = \text{fresh in let } \beta = \text{fresh in}$ $\text{let } r = \text{fresh in let } s = \text{fresh in}$ $\text{let } \theta_3 = unify(T^r \alpha, \theta_2 A_1) \text{ in}$ $\text{let } \theta_4 = unify(T^s \beta, \theta_3 A_2) \text{ in}$ $(\theta_4 \theta_3 \theta_2 \theta_1, T^{\theta_4 \theta_3(r) \odot \theta_4(s)}(\theta_4(\beta)))$
$infer(\Gamma \vdash \text{let } x : S = M_1 \text{ in } M_2 : ?)$	$= \text{let } (\theta_1, A_1) = infer(\Gamma \vdash M_1 : ?) \text{ in}$ $\text{let } (\theta_2, A_2) =$ $infer(\theta_1 \Gamma, x : A_1 \vdash M_2 : ?) \text{ in}$ $\text{let } \alpha = \text{fresh in let } \beta = \text{fresh in}$ $\text{let } \theta_3 = unify(\alpha, \theta_2 A_1) \text{ in}$ $\text{let } \theta_4 = unify(\beta, \theta_3 A_2) \text{ in}$ $(\theta_4 \theta_3 \theta_2 \theta_1, \theta_4(\beta))$

Figure 4.7. Type inference for PolySource.

Theorem 4.3. Assuming $\text{infer}(\Gamma \vdash M : ?) = (\theta, A)$ and $R = \text{fev}(A) \setminus \text{fev}(\Gamma)$, we have that $\theta\Gamma \vdash M : A$ and for any type B such that $\Gamma \vdash M : B$ we have $\forall_{\text{eff}} R(A) > B$.

Proof. These are actually two statements: the returned type is correct and it is the principal type scheme. The former follows by induction on the structure of M , whereas the latter is proved by induction on typing judgements $\Gamma \vdash M : B$. $\square$

4.2.3 Discussion

Polymorphic grades similar to polymorphic effect systems. Hence their capabilities and limitation do not turn out surprising for anybody familiar with effect systems research. In this section we discuss some aspects of extending polymorphic grades.

Recursion Polymorphism and recursion do not go together well. This is seen with traditional polymorphic type schemes [Myc84] and likewise with effects. If we add `fix` M to PolySource, using unification for type inference is no longer possible as it is likely to lead to occurs check problems: for example, r and $r \odot \text{ro}$ cannot be unified.

A possible approach is to do what the analysis-monad library attempts: reintroduce `sub` into the language and use a constraint-based type system such that, whenever an attempt at unification fails, an equality constraint is generated. That way, a recursive computation might have the type $r = r \odot \text{ro} \Rightarrow \text{PermState}_{\text{Int}}^r \text{Int}$.

Smarter monomorphisation The operational semantics of PolySource is defined in terms of a translation into Target. There is an awkward interaction arising from translating typing judgements in a polymorphic language into terms and types in a language without polymorphism, leading to the problem in translating PolySource contexts with type schemes into Target contexts through monomorphisation. The solution in Section 4.2.1 produces all possible monomorphisations of all variables in the context, which does not work for grading algebras with an infinite number of elements. It is also now how monomorphisation works in practical programming languages, as the domain of types is infinite. Instead, we can monomorphise ‘by need’ – that is, only those monomorphic versions of variables needed in the program get instantiated. This would make the translation more complicated, and is hence omitted from the thesis.

4.3 Summary

Analysis-graded monads provide a way of representing program analysis through the facilities provided by graded monadic types in a programming language. If the outcome of the analysis is the grade in the type, then performing the analysis is tantamount to inferring the grade. Under the assumptions placed on analysis-graded monads, we are able to use both constraint solving, seen in classical program analysis and solved by iteration, and unification-based algorithms, as seen in type inference problems for parametrically polymorphic languages. The Source[®] and PolySource languages, seen as simple extensions of the Source language used in all of our examples, exemplify these approaches.

Semantics of analysis-graded monads

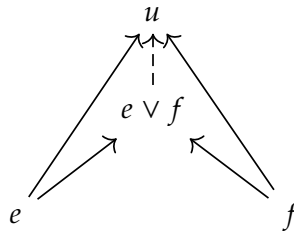
Thus far in the thesis the focus has been on programming with analysis-graded monads. This treatment is incomplete without considering their categorical structure. The aim of this short chapter is just that: exploring the theoretical underpinnings of analysis-graded monads, finding interesting properties relevant for further developments of the theory, and the connection to abstract interpretation.

Chapter structure We first give the categorical definition of analysis-graded monads in Section 5.1 and formalise the terminology for talking about analysis-graded monads. This definition is used to prove important properties of analysis-graded monads, some of which later parts of the thesis rely on. The categorical treatment of analysis-graded monads is then used in Section 5.2 to define the categorical semantics of the Source and Target languages.

5.1 Categorical view

In Chapter 2 I defined graded monads concisely as lax monoidal functors from a monoidal category $\mathcal{M} = (\mathbb{M}, \odot, \mathbf{1})$ to the monoidal category of endofunctors (with composition as the tensor product). Every monoid can be seen as a strict discrete monoidal category whose objects correspond to elements of the monoid carrier set and whose tensor product corresponds to the monoid multiplication operator. In the context of computational grading algebras, we are interested in modelling all their properties using the language of categories.

As with preordered monoids, the grading algebra is a strict thin monoidal category: a morphism $f : x \rightarrow y$ corresponds to the inequality $x \leq y$. Turning the preorder into a partial order means that the existence of morphisms $x \rightarrow y$ and $y \rightarrow x$ implies x and y should be the same – which is the same as saying that isomorphic objects must be equal, so a category representing a pomonoid is also skeletal. The existence of a top element $\top$ means that for every object x there is a unique arrow $x \rightarrow \top$, so $\top$ is a terminal object for the category. Finally, the existence of joins is represented by the fact that, for any two objects e and f , there is an object $e \vee f$ such that, for any other object u such that $e \rightarrow u \leftarrow f$, there is a unique arrow $e \vee f \rightarrow u$ – which trivially means that the following diagram commutes:



This is just the definition of a coproduct.

Let us – for now – omit grade operators from the general notion of computational grading algebras. The categorical definition of analysis-graded monads is then concisely phrased as:

Definition 5.1. A computational grading algebra is a strict skeletal thin monoidal category with a terminal object and all coproducts.

An analysis-graded monad in $\mathbb{C}$ is a lax monoidal functor from a computational grading algebra to the monoidal category of endofunctors on $\mathbb{C}$.

The notion of graded monadic strength is also important (and in practice seen in all monads and graded monads in programming). We thus also consider *strong* analysis-graded monads, which are just strong graded monads that happen to be graded by computational grading algebras.

For the remainder of this section, we explore interesting properties of analysis-graded monads.

Grading algebra operators

The categorical treatment of analysis-graded monads so far omits $\mathcal{O}$. This is because operators in $\mathcal{O}$ do not have require any interesting categorical ‘compatibility’ with other parts of the definition. Interpreting grade operators is required for defining the categorical semantics of Source, so we briefly describe their structure.

A grade operator takes as its arguments (zero or more) elements of the carrier set of the grading algebra. Given that sequencing is a tensor product – meaning a bifunctor – and that one-argument functions are functors, a k -ary grade operator in the grading algebra $\mathcal{G} = (E, \leq, \odot, \mathbf{1})$ is seen as a functor from the k -ary product category $\mathbb{E} \times \cdots \times \mathbb{E}$. That is sufficient, as it already enforces monotonicity in each argument.

5.1.1 Monad at the top

A point emphasised when graded monads were introduced in this thesis was that it is *not* the case that, given a grade e , each T^e is a monad, and the reason for this is that $e \odot e \neq e$ in general. However, there is an interesting property of analysis-graded monads in particular, coming from the following simple fact:

Lemma 5.1. In any computational grading algebra $(E, \leq, \odot, \mathbf{1})$ with top element $\top$, the equality $\top \odot \top = \top$ holds.

Proof. Since $\mathbf{1} \leq \top$, by monotonicity $\top = \top \odot \mathbf{1} \leq \top \odot \top$, so $\top \leq \top \odot \top \leq \top$. By antisymmetry we conclude $\top \odot \top = \top$. $\square$

This follows the intuition that the top element represents the ‘don’t know’, ‘could be anything’ or ‘most general’ outcome of a program analysis. If there is no information about the analysis, then the programmer gets the same amount of information as they would get if they were programming using a non-graded monad. We see this in the definitions of `GMaybe`, `PermStateS` and `LiveStateS`: for the

grade $\top$, the underlying representation of the graded monadic type resembles that of **Maybe** and **State_S**.¹ This result is formalised by the following theorem:

Theorem 5.1. For any graded monad $\mathbf{T} = (T, \mu, \eta) : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$, where $\top \in E$ is the top element of its grading algebra, there is a monad $\mathbf{T}^\top = (T^\top, \mu^\top, \eta^\top)$ in $\mathbb{C}$ defined as follows:

$$\begin{aligned}\mu^\top &= \mu_{\top, \top} \\ \eta^\top &= T(\mathbf{1} \leq \top) \circ \eta\end{aligned}$$

Proof. $T^\top$ is clearly an endofunctor in $\mathbb{C}$ (from the fact $\mathbf{T}$ is a graded monad). The monad associativity and unit laws are just specialisations of their corresponding graded versions, except all grades are $\top$. $\square$

5.1.2 Category of analysis-graded monads

An important component of the formal theory of graded monads is the definition of categories of graded monads. [FKM16] We are interested in the structure and relationship between analysis-graded monads, and it turns out that they form a category – just like monads and graded monads.

Definition 5.2. For a category $\mathbb{C}$, the *analysis-graded monad category* $\mathbf{GMnd}(\mathbb{C})$ is a category with:

objects analysis-graded monads, meaning pairs $(\mathcal{E}, \mathbf{T})$ where $\mathbf{T} = (T, \mu, \eta) : \mathcal{E} \rightarrow \mathbf{Endo}(\mathbb{C})$ is a lax monoidal functor and $\mathcal{E}$ is a small category that is a computational grading algebra;

morphisms $(h, \varphi) : \mathbf{T}_1 \rightarrow \mathbf{T}_2$, where $h : \mathcal{E}_1 \rightarrow \mathcal{E}_2$ is a strict monoidal functor (representing a computational grading algebra homomorphism – we only use the underlying functor by abuse of notation, as the coherence maps are identities), and where φ is a natural transformation $T_1 \Rightarrow T_2$ whose each component φ_e is a natural transformation $T_1 e \Rightarrow T_2(h(e))$ which respects the analysis-graded monadic structure, meaning that for any $e, f \in E_1$ the following diagrams commute:

$$\begin{array}{ccc} T_1 e \bullet T_1 f & \xrightarrow{\varphi_e \bullet \varphi_f} & T_2(h(e)) \bullet T_2(h(f)) \\ (\mu_1)_{e, f} \downarrow & & \downarrow (\mu_2)_{h(e), h(f)} \\ T_1(e \odot_1 f) & \xrightarrow{\varphi_{e \odot_1 f}} T_2(h(e \odot_1 f)) & = T_2(h(e) \odot_2 h(f)) \end{array}$$

$$\begin{array}{ccc} \text{Id}_{\mathbb{C}} & \xrightarrow{\eta_1} & T_1 \mathbf{1}_1 \\ & \searrow \eta_2 & \downarrow \varphi_{\mathbf{1}_1} \\ & & T_2 \mathbf{1}_2 \end{array}$$

¹The **HistState_S** graded monad notably does *not* have the type constructor for grade $\top = \omega$ match the one for the **State_S** monad. However, the type constructor **HistState_S^ω**, mapping types A to $S \rightarrow A \times \mathbf{List} S$, also admits a monadic structure.

composition for $(h, \varphi) : \mathbf{T}_1 \rightarrow \mathbf{T}_2$ and $(h', \varphi') : \mathbf{T}_2 \rightarrow \mathbf{T}_3$, define $(h', \varphi') \circ (h, \varphi) = (h' \bullet h, \varphi' \star \varphi)$, where the natural transformation $\varphi' \star \varphi$ is defined with $(\varphi' \star \varphi)_e = \varphi'_{h_1(e)} \circ \varphi_e$ for all $e \in E_1$;

identities for each graded monad $\mathbf{T} = (T, \mu, \eta) : \mathcal{E} \rightarrow \mathbf{Endo}(\mathbb{C})$, the identity morphism is $\text{id}_{\mathbf{T}} = (\text{id}_{\mathcal{E}}, \text{id})$.

It can be shown that this definition is valid (meaning that composition is associative and that the identity, as defined, is really the identity).

In practice, we also care about the subcategory of strong analysis-graded monads (T, μ, η, t) on $\mathbb{C}$ with tensorial strength t preserved by morphisms, which we call $\mathbf{SGMnd}(\mathbb{C})$ – simply put, all non-strong monads are omitted.

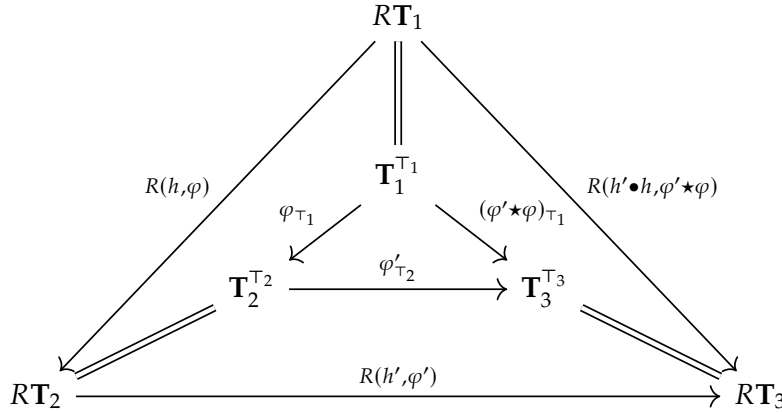
A simple result formalises the intuition that analysis-graded monadic computations ‘elaborate’ those represented by monads. This relationship means that there exists an adjoint pair of functors between $\mathbf{Mnd}(\mathbb{C})$ and $\mathbf{GMnd}(\mathbb{C})$.

Theorem 5.2. The functors $R : \mathbf{Mnd}(\mathbb{C}) \rightarrow \mathbf{GMnd}(\mathbb{C})$ and $L : \mathbf{GMnd}(\mathbb{C}) \rightarrow \mathbf{Mnd}(\mathbb{C})$, defined with:

$$\begin{aligned} R(T, \mu, \eta) &= \widehat{\mathbf{T}} = (\widehat{T}, \widehat{\mu}, \widehat{\eta}) : \mathbb{1} \rightarrow \mathbf{Endo}(\mathbb{C}) & R\varphi &= (\text{id}, \varphi) \\ \text{where: } \widehat{T} &= T, \quad \widehat{\mu}_{*,*} = \mu, \quad \widehat{\eta} = \eta \\ L((T, \mu, \eta) : \mathcal{E} \rightarrow \mathbf{Endo}(\mathbb{C})) &= \widehat{\mathbf{T}} = (T^\top, \mu^\top, \eta^\top) & L(h, \varphi) &= \varphi_\top \\ \text{where: } \mu^\top &= \mu_{\top, \top}, \quad \eta^\top = T(\mathbf{1} \leq \top) \end{aligned}$$

are adjoint: $L \dashv R$.

Proof. Both L and R are easily shown to be functors – R is a functor by Theorem 5.1 and because the following diagram commutes (showing preservation of composition):



To show these functors are adjoint, first note that for any monad $\mathbf{T} = (T, \mu, \eta)$ we have $LRT = \mathbf{T}$, so the counit of the adjunction is the identity natural transformation (in $\mathbb{C}$): $\text{id} : LR \Rightarrow \text{id}_{\mathbb{C}}$. Next, for a graded monad $\mathbf{T}$, $RL\mathbf{T}$ is a $\mathbb{1}$ -graded monad, meaning the only grade is $*$, and the multiplication and unit for this graded monad match the ones for $\mathbf{T}^\top$. Therefore, the unit of the adjunction is the natural transformation $\widehat{\mathbf{T}} : \text{id}_{\mathbb{C}} \rightarrow RL$ such that for all $\mathbf{T} \in \mathbf{GMnd}(\mathbb{C})$:

$$\widehat{\mathbf{T}} = (h, \varphi) \quad \text{where:} \quad h(e) = \top, \quad \varphi_e = T(e \leq \top) \quad \text{for } e \in E$$

It is trivially shown that both of the universal properties of unit and counit of the adjunction hold. $\square$

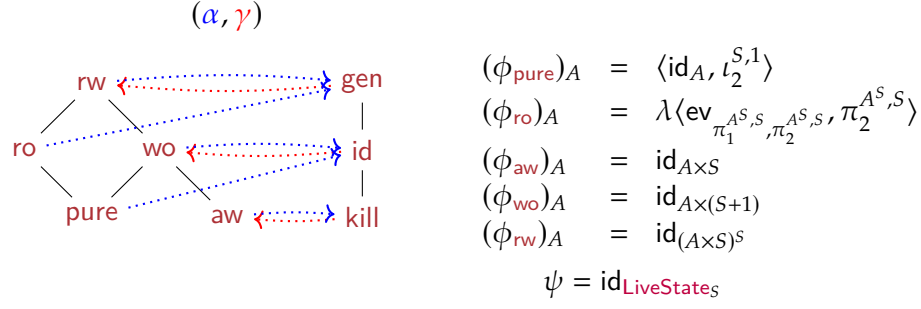


Figure 5.1. LiveState_S is an abstraction of PermState_S .

5.1.3 Abstractions

Computational grading algebras are inspired by the theory of abstract interpretation. Thus far we have not considered how their definition and that of analysis-graded monads relates to properties of abstract interpretation – in particular the key idea of abstraction and concretisation functions. Exploring these questions is the aim of this section.

First consider the PermState_S and LiveState_S graded monads. We can think of LiveState_S as a ‘simplified’ version of PermState_S : the computational grading algebra has fewer elements and there are clearly matching translations for some graded monadic types. For example, $\langle\langle \text{PermState}_S^{\text{wo}} A \rangle\rangle_{\text{type}} = \langle\langle \text{LiveState}_S^{\text{id}} A \rangle\rangle_{\text{type}}$. This correspondence makes sense: live variable analysis is meant to infer whether a variable is definitely written to before being read from, whereas permission-based analysis covers a richer set of possible outcomes. Thus LiveState_S is an *abstraction* of PermState_S .

We now formally define the notion of abstraction for graded monads:

Definition 5.3. A graded monad $\mathbf{T}_2 : \mathcal{E}_2 \rightarrow \mathbf{Endo}(\mathbb{C})$ is an *abstraction* of graded monad $\mathbf{T}_1 : \mathcal{E}_1 \rightarrow \mathbf{Endo}(\mathbb{C})$ if there exists a tuple $(\alpha, \gamma, \phi, \psi)$ such that:

- $(\alpha, \phi) : \mathbf{T}_1 \rightarrow \mathbf{T}_2$ and $(\gamma, \psi) : \mathbf{T}_2 \rightarrow \mathbf{T}_1$ are morphisms in $\mathbf{GMnd}(\mathbb{C})$;
- (α, γ) is a abstraction-concretisation (Galois connection) pair – for all $e_1 \in E_1$, $e_1 \leq \gamma(\alpha(e_1))$, and for all $e_2 \in E_2$, $\alpha(\gamma(e_2)) \leq e_2$;
- composing morphisms is the same as subeffecting, meaning that for any $e \in E_1$:

$$\psi_{\alpha(e)} \circ \phi_e = T_1(e \leq_1 \gamma(\alpha(e)))$$

It is easily shown that LiveState_S is an abstraction of PermState_S . The corresponding abstraction-concretisation pair, as well as the $\mathbf{GMnd}(\mathbb{C})$ morphisms, are given in Figure 5.1.

The formula definition of graded monadic abstractions matches the intuitive understanding. Suppose $\mathbf{T}_1$ represents some effectful computations and is capable of representing many possible analysis outcomes, whereas $\mathbf{T}_2$ represents the same kinds of computations, but simplifies the analysis. It might be the case that that checking equality for the grading algebra $\mathcal{E}_1$ is undecidable, which makes type checking difficult, but everything in $\mathcal{E}_2$ is decidable. We mentioned such examples in Chapter 2.

To emphasise this point, we have the following result relating $\mathbf{T}$ and $\mathbf{T}^\top$:

Lemma 5.2. For any graded monad $\mathbf{T}$, the $\mathbb{1}$ -graded monad corresponding to the monad $\mathbf{T}^\top$ is an abstraction of $\mathbf{T}$.

Proof. The abstraction is achieved with $(\alpha, \gamma, \phi, \psi)$, where $\alpha(e) = *$, $\gamma(*) = \top$, $\phi_e = T(e \leq \top)$ and $\psi_* = T * \text{id}$. $\square$

5.2 Categorical semantics

The categorical treatment of analysis-graded monads in Section 5.1 has two main uses: investigating their formal properties; providing denotational semantics of the Source and Target languages from Chapter 3. We now turn to the latter.

We first define the categorical semantics of Target, which is simple because the language is just the simply-typed lambda calculus with product and sum types. While we could leave it at that, we review the necessary machinery in order to review the conventions and notation that are then used in Source. Afterwards, we define the categorical semantics of Source, which relies on the categorical view of Target. One goal of defining the semantics is showing that translation from Source to Target preserves denotational semantics, a result which we formally state.

5.2.1 Semantics of Target

The Target language is a variant of simply-typed lambda calculus (STLC) with sums and products. The semantics of STLC is given in terms of a cartesian-closed category (ccc) $\mathbb{C}$, meaning a category with products, exponentials and a terminal object 1 . The key feature of Target that places a stronger requirement on $\mathbb{C}$ is the presence of sum types, which are typically interpreted as coproducts. The product and coproduct should satisfy a distributivity law, meaning that the canonical map $(A \times B) + (A \times C) \rightarrow A \times (B + C)$ is an isomorphism for all objects A, B and C . Hence for the remainder of this section we assume that the category of discourse $\mathbb{C}$ is a ccc with binary coproducts – which implies the distributivity property. The segment of Target without sums does not require finite coproducts, so excluding analysis-graded monads using sums such as **GMaybe** from consideration relaxes the conditions on $\mathbb{C}$.

Furthermore, the variant of Target with the empty type **Emp** (used in Chapter 4) requires the existence of the initial object 0 in $\mathbb{C}$. In our presentation of Target we do not assume an initial object, but the modification is trivial. Note that the requirement that $\mathbb{C}$ is a ccc, has an initial object, and has all binary coproducts implies the category is distributive.

The semantics of Target in $\mathbb{C}$ is stated in terms of three interpretation functions:

types the function $\llbracket \cdot \rrbracket_{\text{type}}$ maps Target types A to objects of $\mathbb{C}$;

environments the function $\llbracket \cdot \rrbracket_{\text{v2t}}$ maps Target environments Γ to objects of $\mathbb{C}$ according to the following definition:

$$\begin{aligned} \llbracket \diamond \rrbracket_{\text{v2t}} &= 1 \\ \llbracket x_1 : A_1, \dots, x_n : A_n \rrbracket_{\text{v2t}} &= \llbracket A_1 \rrbracket_{\text{type}} \times \dots \times \llbracket A_n \rrbracket_{\text{type}} \end{aligned}$$

typing judgements the function $\llbracket \cdot \rrbracket_{\text{val}}$ maps Target typing judgements $\Gamma \vdash M : A$ to morphisms $\llbracket \Gamma \rrbracket_{\text{v2t}} \rightarrow \llbracket A \rrbracket_{\text{type}}$.

The full definitions of semantic functions $\llbracket \cdot \rrbracket_{\text{type}}$ and $\llbracket \cdot \rrbracket_{\text{val}}$ rely on some given denotations of the type `Int`, typing judgements for its constants n and binary operator terms $M \text{ op } N$. Any further base types B and their constants c require the same. Other than that, the $\llbracket \cdot \rrbracket_{\text{type}}$ and $\llbracket \cdot \rrbracket_{\text{val}}$ definitions are given in Figure 5.2. Since the definitions of semantics of judgements rely on their derivations, we adapt the notation from Chapter 3 used for translations in cases where the derivation is ambiguous (expect we use $m_1, \dots, m_k$ instead $t_1, \dots, t_k$). These are not far from what we expect, but we briefly look at two of them.

From STLC we have that the semantics of function abstraction corresponds to the unique morphism $\lambda \llbracket \Gamma, x : A \vdash M : B \rrbracket_{\text{v2t}} : \llbracket \Gamma \rrbracket_{\text{v2t}} \rightarrow \llbracket B \rrbracket_{\text{type}}^{\llbracket A \rrbracket_{\text{type}}}$ such that the following diagram commutes:

$$\begin{array}{ccc}
 \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A \rrbracket_{\text{type}} & \xrightarrow{\lambda \llbracket \Gamma \vdash M : B \rrbracket_{\text{val}} \times \text{id}_{\llbracket A \rrbracket_{\text{type}}}} & \llbracket B \rrbracket_{\text{type}}^{\llbracket A \rrbracket_{\text{type}}} \times \llbracket A \rrbracket_{\text{type}} \\
 \parallel & & \downarrow \text{ev}_{\llbracket A \rrbracket_{\text{type}}, \llbracket B \rrbracket_{\text{type}}} \\
 \llbracket \Gamma, x : A \rrbracket_{\text{v2t}} & \xrightarrow{\llbracket \Gamma, x : A \vdash M : B \rrbracket_{\text{val}}} & \llbracket B \rrbracket_{\text{type}}
 \end{array}$$

On the other hand, the interpretation of the `case` construct is more sophisticated and is based on the following diagram (which commutes by the universal property of coproducts):

$$\begin{array}{ccccc}
 & & \llbracket \Gamma \rrbracket_{\text{v2t}} & & \\
 & & \downarrow \langle \text{id}_{\llbracket \Gamma \rrbracket_{\text{v2t}}}, \llbracket \Gamma \vdash M : A_1 + A_2 \rrbracket_{\text{val}} \rangle & & \\
 & & \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A_1 + A_2 \rrbracket_{\text{type}} & & \\
 & & \parallel & & \\
 & \swarrow & & \searrow & \\
 \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A_1 \rrbracket_{\text{type}} & & \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A_1 \rrbracket_{\text{type}} + \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A_2 \rrbracket_{\text{type}} & & \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A_2 \rrbracket_{\text{type}} \\
 \uparrow \iota_1 & & & & \uparrow \iota_2 \\
 \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A_1 \rrbracket_{\text{type}} & & & & \llbracket \Gamma \rrbracket_{\text{v2t}} \times \llbracket A_2 \rrbracket_{\text{type}} \\
 \parallel & & & & \parallel \\
 \llbracket \Gamma, x_1 : A_1 \rrbracket_{\text{v2t}} & & & & \llbracket \Gamma, x_2 : A_2 \rrbracket_{\text{v2t}} \\
 \swarrow m_1 = \llbracket \Gamma, x_1 : A_1 \vdash N_1 : B \rrbracket_{\text{val}} & & \downarrow [m_1, m_2] & & \swarrow \llbracket \Gamma, x_2 : A_2 \vdash N_2 : B \rrbracket_{\text{val}} = m_2 \\
 & & \llbracket B \rrbracket_{\text{type}} & &
 \end{array}$$

With the categorical semantics defined, the following *soundness* property holds:

Theorem 5.3. In a category $\mathbb{C}$, for any two terms $M \rightsquigarrow N$ with $\Gamma \vdash M : A$ (and hence $\Gamma \vdash N : A$):

$$\llbracket \Gamma \vdash M : A \rrbracket_{\text{val}} = \llbracket \Gamma \vdash N : A \rrbracket_{\text{val}}$$

Proof. By induction on derivations of $M \rightsquigarrow N$. □

The semantics of Target is not particularly interesting in the context of analysis-graded monads. However, its formal definition matters for the sake of showing that the categorical semantics of Source terms and their translations into Target match.

$$\begin{array}{ll}
\llbracket \text{Unit} \rrbracket_{\text{type}} &= 1 \\
\llbracket A \rightarrow B \rrbracket_{\text{type}} &= \llbracket B \rrbracket_{\text{type}}^{\llbracket A \rrbracket_{\text{type}}} \\
\llbracket A \times B \rrbracket_{\text{type}} &= \llbracket A \rrbracket_{\text{type}} \times \llbracket B \rrbracket_{\text{type}} \\
\llbracket A + B \rrbracket_{\text{type}} &= \llbracket A \rrbracket_{\text{type}} + \llbracket B \rrbracket_{\text{type}}
\end{array}$$

$$\begin{array}{ll}
\llbracket \Gamma \vdash \langle \rangle : \text{Unit} \rrbracket_{\text{val}} &= !\llbracket \Gamma \rrbracket_{\text{v2t}} \\
\llbracket x_1 : A_1, \dots, x_n : A_n \vdash x_i : A_i \rrbracket_{\text{val}} &= \pi_i^{A_1, \dots, A_n} \\
\llbracket \Gamma \vdash \text{fn } x : A \Rightarrow M : A \rightarrow B \rrbracket_{\text{val}} &= \lambda \llbracket \Gamma, x : A \vdash M : B \rrbracket_{\text{val}}
\end{array}$$

$$\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\llbracket \Gamma \vdash M N : B \rrbracket_{\text{val}}} \quad \bullet \quad \text{ev}_{A,B} \circ \langle m_1, m_2 \rangle$$

$$\llbracket \Gamma \vdash \langle M_1, \dots, M_n \rangle : A_1 \times \dots \times A_n \rrbracket_{\text{val}} = \langle \llbracket \Gamma \vdash M_1 : A_1 \rrbracket_{\text{val}}, \dots, \llbracket \Gamma \vdash M_n : A_n \rrbracket_{\text{val}} \rangle$$

$$\frac{\Gamma \vdash M : A_1 \times \dots \times A_n}{\Gamma \vdash \text{proj}_i M : A_i} \quad \bullet \quad \pi_i^{A_1, \dots, A_n} \circ m_1$$

$$\begin{array}{ll}
\llbracket \Gamma \vdash \text{inl } M : A + B : A + B \rrbracket_{\text{val}} &= \iota_1^{A,B} \circ \llbracket \Gamma \vdash M : A \rrbracket_{\text{val}} \\
\llbracket \Gamma \vdash \text{inr } M : A + B : A + B \rrbracket_{\text{val}} &= \iota_2^{A,B} \circ \llbracket \Gamma \vdash M : B \rrbracket_{\text{val}}
\end{array}$$

$$\frac{\Gamma \vdash M : A_1 + A_2 \quad \Gamma, x_1 : A_1 \vdash N_1 : B \quad \Gamma, x_2 : A_2 \vdash N_2 : B}{\Gamma \vdash \text{case } M \text{ of } (\text{inl } x_1 \Rightarrow N_1 \mid \text{inr } x_2 \Rightarrow N_2) : B} = [m_2, m_3] \circ \langle \text{id}_{\llbracket \Gamma \rrbracket_{\text{v2t}}}, m_1 \rangle$$

Figure 5.2. Categorical semantics of Target (omitting **Int**).

$$\begin{aligned}
\llbracket \text{Unit} \rrbracket_{\text{type}} &= 1 & \llbracket T^e A \rrbracket_{\text{type}} &= T \llbracket e \rrbracket_{\text{gr}} \llbracket A \rrbracket_{\text{type}} \\
\llbracket A \rightarrow B \rrbracket_{\text{type}} &= \llbracket B \rrbracket_{\text{type}}^{\llbracket A \rrbracket_{\text{type}}} \\
\llbracket \Gamma \vdash \langle \rangle : \text{Unit} \rrbracket_{\text{val}} &= !\llbracket \Gamma \rrbracket_{\text{v2t}} \\
\llbracket x_1 : A_1, \dots, x_n : A_n \vdash x_i : A_i \rrbracket_{\text{val}} &= \pi_i^{A_1, \dots, A_n} \\
\llbracket \Gamma \vdash \text{fn } x : A \Rightarrow M : A \rightarrow B \rrbracket_{\text{val}} &= \lambda \llbracket \Gamma, x : A \vdash M : B \rrbracket_{\text{val}} \\
\frac{\Gamma \vdash M : A \rightarrow B \quad \Gamma \vdash N : A}{\Gamma \vdash M N : B} &\quad \bullet \quad \sim \quad \text{ev}_{\llbracket A \rrbracket_{\text{type}}, \llbracket B \rrbracket_{\text{type}}} \circ \langle m_1, m_2 \rangle \\
\llbracket \Gamma \vdash \text{return } M : T^i A \rrbracket_{\text{val}} &= \eta_{\llbracket A \rrbracket_{\text{type}}} \circ \llbracket \Gamma \vdash M : A \rrbracket_{\text{val}} \quad (\text{where } i = 1) \\
\frac{\Gamma \vdash M : T^e A}{\Gamma \vdash \text{sub}_{e,f} M : T^f A} &\quad \bullet \quad \sim \quad T(\llbracket e \rrbracket_{\text{gr}} \leq \llbracket f \rrbracket_{\text{gr}}) \llbracket A \rrbracket_{\text{type}} \circ m_1 \\
\frac{\Gamma \vdash M : T^e A \quad \Gamma, x : A \vdash N : T^f A}{\Gamma \vdash \text{bind } x : A \leftarrow M \text{ with } N : T^g A} &= \quad (\text{where } \vdash_{\text{gr}} e \odot f = g) \\
&\quad \mu_{e,f} \circ T \llbracket e \rrbracket_{\text{gr}} m_2 \circ (t_e)_{\llbracket \Gamma \rrbracket_{\text{v2t}}, \llbracket A \rrbracket_{\text{type}}} \circ \langle \text{id}_{\llbracket \Gamma \rrbracket_{\text{v2t}}}, m_1 \rangle
\end{aligned}$$

Figure 5.3. Categorical semantics of Source (excluding **Int**), assuming the underlying strong graded monad $\mathbf{T} = (T, \mu, \eta, t) : \mathcal{E} \rightarrow \mathbf{Endo}(\mathbb{C})$.

5.2.2 Semantics of Source

The categorical semantics of Source relies on a particular strong graded monad $\mathbf{T} = (T, \mu, \eta, t) : (E, \leq, \odot, 1) \rightarrow \mathbf{Endo}(\mathbb{C})$. In addition to $\llbracket \cdot \rrbracket_{\text{val}}$, $\llbracket \cdot \rrbracket_{\text{type}}$ and $\llbracket \cdot \rrbracket_{\text{v2t}}$ like in Target, the semantics of Source relies on the semantics of grades $\llbracket \cdot \rrbracket_{\text{gr}}$ and of graded monads $\llbracket \cdot \rrbracket_{\text{gm}}$ – what object in the grading algebra category $\mathcal{E}$ does a grade correspond to and what graded monad in a graded monadic type corresponds to. We slightly abuse notation and use the letter T to refer to both the graded monad in the type $T^e A$ and the functor associated with the analysis-graded monad $\mathbf{T}$ and thus avoid writing $\llbracket T \rrbracket_{\text{gm}} = \mathbf{T}$ (and then further writing $\llbracket T \rrbracket_{\text{gm}}^T$ to extract the functor or $\llbracket T \rrbracket_{\text{gm}}^\mu$ for multiplication). This allows us to define the semantics of grades and graded monads simply as:

$$\llbracket e \rrbracket_{\text{gr}} = e, \quad \llbracket T \rrbracket_{\text{gm}} = \mathbf{T} = (T, \mu, \eta, t)$$

The semantics of Source is shown in Figure 5.3. It resembles the semantics given by Gaboardi et al. [GKO⁺16] in their language featuring programming with effects and coeffects, though in their work the programmer does not use graded monadic types to represent side-effecting computations. It also resembles the semantics of programming with monads.

The most involved part of the definition of the categorical semantics concerns the semantics of binding. It is more useful to consider each step of the translation:

$$\begin{array}{ccc}
 \begin{array}{c} \llbracket \Gamma \rrbracket_{v2t} \\ \downarrow \langle \text{id} \rrbracket_{v2t}, \llbracket \Gamma \vdash M : T^e A \rrbracket_{val} \rangle \\ \llbracket \Gamma \rrbracket_{v2t} \times Te \llbracket A \rrbracket_{type} \\ \downarrow (t_e) \llbracket \Gamma \rrbracket_{v2t}, \llbracket A \rrbracket_{type} \rangle \\ Te(\llbracket \Gamma \rrbracket_{v2t} \times \llbracket A \rrbracket_{type}) \end{array} & \begin{array}{c} Te(\llbracket \Gamma, x : A \rrbracket_{v2t}) \\ \nearrow Te(\llbracket \Gamma, x : A \vdash N : T^f B \rrbracket_{val}) \\ Te(Tf \llbracket B \rrbracket_{type}) \\ \downarrow (\mu_e, f) \llbracket B \rrbracket_{type} \\ T(e \odot f) \llbracket B \rrbracket_{type} \\ \parallel \\ Tg \llbracket B \rrbracket_{type} \end{array}
 \end{array}$$

One thing that is *not* included in this general treatment is the categorical semantics of all of the constructs specific to a given graded monad, for example `get` for graded monads representing stateful computation and `nothing` for `GMaybe`. These are not difficult to define, so for the sake of an example we give the categorical semantics of `get` and `put` for `PermStateS`:

$$\begin{aligned}
 \llbracket \Gamma \vdash \text{get} : \text{PermState}_S^{\text{ro}} S \rrbracket_{val} &= \lambda \langle \text{id} \rrbracket_{v2t}, \text{id} \rrbracket_{S \rrbracket_{type}} \rangle \\
 \llbracket \Gamma \vdash \text{put } M : \text{PermState}_S^{\text{aw}} \text{Unit} \rrbracket_{val} &= \langle \text{id} \rrbracket_{v2t}, \llbracket \Gamma \vdash M : S \rrbracket_{val} \rangle
 \end{aligned}$$

Simply put, the semantics of `get` represents the fact that the underlying representation of `PermStateSro S` is $\llbracket S \rrbracket_{type}^{\llbracket S \rrbracket_{type}}$ and that `get` is a trivial identity function on S , whereas `put` is tied to the fact that `PermStateSaw Unit` is represented as $\text{Unit} \times S$, with the first coordinate always $\langle \rangle$ and the second coordinate matching the semantics of M .

Translation preserves semantics

While for Target we have a soundness result relating the operational and categorical semantics, we do not have the equivalent result for Source because the operational semantics of Source is not defined. Instead, Target is a kernel language that Source terms and types get translated into, meaning that the operational semantics of Source is considered only in terms of Target. Therefore, the closest result corresponding to soundness is the fact that *the translation preserves semantics*.

This result is conditional on the translation itself being ‘sensible’, meaning that the representation of Source graded monadic types in Target matches the corresponding object in category theory. Likewise, all operations dealing with graded monadic types have to be translated exactly corresponding to their semantics.

Theorem 5.4. Suppose that a translation from Source to Target satisfies the following properties:

- for any Source type A and grade e :

$$\llbracket \llbracket T^e A \rrbracket_{type} \rrbracket_{type} = T \llbracket e \rrbracket_{gr} \llbracket \llbracket A \rrbracket_{type} \rrbracket_{type};$$

- all of the graded monadic terms have similarly matching denotations.

Then the following semantics preservation properties of translation holds:

1. for any Source type A : $\llbracket A \rrbracket_{type} = \llbracket \llbracket A \rrbracket_{type} \rrbracket_{type};$

2. for any Source environment Γ : $\llbracket \Gamma \rrbracket_{\text{v2t}} = \llbracket \langle \Gamma \rangle_{\text{env}} \rrbracket_{\text{v2t}}$;
3. for any derivable Source judgement $\Gamma \vdash M : A$, we have:

$$\llbracket \Gamma \vdash M : A \rrbracket_{\text{val}} = \llbracket \langle \Gamma \rangle_{\text{env}} \vdash \langle M \rangle_{\text{val}} : \langle A \rangle_{\text{type}} \rrbracket_{\text{val}}.$$

Proof. By simple induction and case distinction:

1. By induction on structure of types in Source. A lot of the ‘heavy lifting’ is done by the assumption of the theorem.
2. Immediately by definition of $\langle \cdot \rangle_{\text{env}}$ and $\llbracket \cdot \rrbracket_{\text{v2t}}$.
3. By induction on derivations of $\Gamma \vdash M : A$ judgements. All of the cases dealing with graded monads are handled by the second assumption, whereas the remaining ones are simple.

□

While the result is simple, it is useful to reflect on how the categorical semantics defined in this chapter matches with the approach to semantics of Source given in Chapter 3. Earlier, we considered a purely operational view of graded monadic types, seeing them as merely aliases for some underlying data structure. The denotational (categorical) approach of this chapter relates the analysis-graded monads seen in programming to analysis-graded monads defined categorically. Provided the ‘blanks’ in the operational approach are ‘filled correctly’, these two approaches match – as is expected.

Note on abstract interpretation

In section 5.1.3, we defined what it means for an analysis-graded monad $\mathbf{T}_2$ to be an abstraction of $\mathbf{T}_1$, relying on the idea of abstract interpretation by Cousot and extending it from the computational grading algebra to the analysis-graded monad itself. We defined the semantics of Source, so it is natural to ask how this idea is relevant in a programming context – which is why we started this exercise.

While Source, as presented, only supports the use of one graded monad $\mathbf{T}$, we can temporarily forget that this is the case and use two graded monads $\mathbf{T}_1$ and $\mathbf{T}_2$. Suppose $\mathbf{T}_2$ is an abstraction of $\mathbf{T}_1$ in the sense defined in section 5.1.3. This means that a term whose type is $T_2^{a(e)} \text{Int}$ has a less precise (or less restricted) representation than one whose type is $T_1^e \text{Int}$. Abstract interpretation comes at the cost of loss of precision in an analysis. This is seen from the following fact.

For an abstraction witness $(\alpha, \gamma, \phi, \psi)$, performing abstraction and concretisation gives the same result as subeffecting – in other words, for any derivable $\Gamma \vdash M : T_1^e A$:

$$\llbracket \Gamma \vdash \text{sub}_{e, \gamma(\alpha(e))} M \rrbracket_{\text{val}} = (\psi_e)_{\llbracket A \rrbracket_{\text{type}}} \circ (\phi_e)_{\llbracket A \rrbracket_{\text{type}}} \circ \llbracket \Gamma \vdash M : T_1^e A \rrbracket_{\text{val}}$$

This equality holds – the following diagram commutes by definition of abstraction and semantics of subeffecting:

$$\begin{array}{ccc}
 \llbracket \Gamma \rrbracket_{\text{v2t}} & & \\
 \downarrow \llbracket \Gamma \vdash M : T_1^e A \rrbracket_{\text{val}} & \searrow \llbracket \Gamma \vdash \text{sub}_{e, \gamma(\alpha(e))} M \rrbracket_{\text{val}} & \\
 T_1 e \llbracket A \rrbracket_{\text{type}} & \xrightarrow{(T_1(e \leq_1 f)) \llbracket A \rrbracket_{\text{type}}} & T_1(\gamma(\alpha(e)) \llbracket A \rrbracket_{\text{type}}) \\
 \downarrow (\phi_e) \llbracket A \rrbracket_{\text{type}} & & \nearrow (\psi_e) \llbracket A \rrbracket_{\text{type}} \\
 T_2(\alpha(e)) \llbracket A \rrbracket_{\text{type}} & &
 \end{array}$$

Furthermore, this equality can be interpreted as an analogue of the ‘valid abstraction’ part of the definition of abstract interpretation (Definition 2.8). The semantics of M (denoted f in the abstract interpretation context) is the morphism $\llbracket \Gamma \vdash M : T_1^e A \rrbracket_{\text{val}}$, the abstract semantics ($\bar{f}$) is $(\phi_e) \llbracket A \rrbracket_{\text{type}} \circ \llbracket \Gamma \vdash M : T_1^e A \rrbracket_{\text{val}}$, contextual abstraction is the identity and concretisation is given by $(\psi_e) \llbracket A \rrbracket_{\text{type}}$.

5.3 Summary

In this chapter, we considered the categorical treatment of analysis-graded monads. Adding a top element to the order allows us to make a connection between monads and graded monads. Furthermore, we define the categorical semantics of the Source language in terms of analysis-graded monads in category theory. We also show that the translation from Source to Target given in Chapter 3 preserves categorical semantics – giving a soundness of translation result. Along the way we explore the relationship between analysis-graded monads and program analysis by considering the question of abstraction. We show that there is a sense in which a graded monad represents an abstraction of an analysis represented by another graded monad – both in the foundational mathematics sense and in the more commonly understood programming sense.

Part III

Adding structure

Part II was mainly the introduction of a framework for programming with analysis-graded monads, both from a practical and from a fundamental perspective. Those chapters exposed limitations of programming with analysis-graded monads in settings that go beyond one analysis for one computation. We now address some of these limitations.

Chapter 6 considers the problem of programming with and easily defining analysis-graded monads that have multiple ‘components’ (such as different exception or mutable variable names). The solution is the use of *graded monad builders*, which are a proposed syntax that concisely defines such graded monads when supplied the set of names of these components (called *symbols* in this work).

Chapter 7 turns to the question of using multiple analyses at the same time, meaning that it addresses the issue of combining the functionality of several graded monads at the same time. We consider two such combination operations. Combining two side-effecting computations is achieved using *graded monad transformers*, a mostly simple generalisation of the non-graded concept familiar to functional programmers. The question of combining two analysis for computations based on the *same* side-effect is also addressed, and a solution in the form of *grade families* is suggested.

Graded monad builders

In Chapter 3, we looked at various graded monads useful for program analysis and discussed their implementations in Haskell, as an example of a practical, general-purpose functional programming language. While the examples of exceptions, simple structured locking and stateful computation can all be written without much difficulty, we already faced the issue of having to write a significant amount of boilerplate. We identified several variants of this problem, including writing duplicate code for almost identical grading algebras when implementing the graded monad in Haskell.

The grading algebras we considered are parameterised by a set Σ whose interpretation depends on the application: set of exceptions, set of mutable variables and so on. Then, for example, for each exception $E \in \Sigma$ we have a separate `raiseE` graded monadic computation. In this chapter we present an alternative approach, where there is only a single `raise` function that takes an exception name as an argument. This is more in line with what we are used to seeing in conventional programming languages (for example, exception handling in OCaml). However, now the argument (exception name) appears in the type signature of `raise`, similarly to dependently-typed programming languages. Thus, for example, the type of `raise E1` is `Exception{E1} α` and the type of `raise E2` is `Exception{E2} α`. We manage to find a sweet spot: a language that supports such types, yet it features decidable type inference.

The way we achieve this functionality is by introducing the notion of *symbols*, which are a separate kind from values, types and effects. These are the elements of Σ , and can be used to construct other kinds. For example, individual exception names (e.g. `InvalidIndex`) are symbols, and the grading algebra is parameterised by the set of symbols (with the underlying carrier set $\mathcal{P}(\Sigma)$). A *set of symbols* can then be given as an argument to a *graded monad builder*, which performs the role of generating all necessary definitions of the grading algebra and the graded monadic operations needed for programming with this side-effect and analysis.

Chapter structure In this chapter, I define and describe two calculi for lexically scoped symbols: ML_G and System G. The former is a functional language with parametric polymorphism in effects and types, extending Hindley-Milner-Damas polymorphism seen in ML and other functional languages. The latter is similar to System F, but extended with polymorphic effects and abstractions over symbols. ML_G admits a decidable type and effect inference algorithm, meaning that performing a decidable program analysis reduces to effect inference.

In Section 6.1, I discuss the ways symbols might be used in a practical programming language, as well as motivate (and preemptively justify) design decisions made in ML_G and System G. I also show where lexical scoping comes in and its relevance

to this task. Next, I turn to System G in Section 6.2, where I define the calculus and provide its denotational semantics. Finally, I define ML_G in Section 6.3, as well as show examples written in ML_G and explain how the type inference algorithm works.

Relevant publications The contents of this chapter are based on an unpublished manuscript I co-authored with Alan Mycroft and Neel Krishnaswami, whose current title is *Graded monad builders*.

6.1 Towards builders and symbols

In this section we consider a few examples of analyses expressed using graded monad builders, as well as the idea of symbol-parameterised grading algebras. We start the section with examples, and then give a few quick definitions in order to make the language consistent. All of this is done to motivate the design decisions made in ML_G and System G (using syntax seen in ML_G) before the languages are formally defined.

6.1.1 Examples and syntax

Consider again permission-graded stateful computations. Suppose that the monadic bind operator in our language is implemented using `bind  $x \Leftarrow M_1$  with  $M_2$` syntax (equivalent to $M_1 \gg= \lambda x. M_2$ elsewhere). If the language has a kind of symbols (here X and Y), then we write code such as the following in ML_G :

```
fn a  $\Rightarrow$  (bind  $x \Leftarrow$  get  $X$  with (bind  $y \Leftarrow$  get  $Y$  with put  $Y$  ( $a + x + y$ ))),
```

and the type of this expression would be $\text{Int} \rightarrow \text{PermState}_{\{\{X,Y\}\}}^{\{X \mapsto \text{ro}, Y \mapsto \text{rw}\}} \text{Unit}$, provided symbols X and Y are in scope. The type constructor $\text{PermState}_{\{\{X,Y\}\}}^F$ for a grade F has a subscript that marks what the underlying symbol set is – essentially instantiating the graded monad builder PermState . In our system names of kind symbol may be defined in a top-level way, maybe using syntax such as:

```
symbols  $X, Y$ 
```

We can parameterise the above function on its symbol arguments (here V, W) rather than fixing X and Y ahead of time. We can then write the following:

```
sym  $V \Rightarrow$  sym  $W \Rightarrow$  fn a  $\Rightarrow$  (bind  $x \Leftarrow$  get  $V$  with  
(bind  $y \Leftarrow$  get  $W$  with put  $W$  ( $a + x + y$ ))).
```

The type of this expression is no longer a simple function type: the names V and W , which appear as arguments, also appear in the type. We borrow notation from dependently-typed languages and use the notation $\Pi_{\text{sym}} \delta (A)$ to represent such types. The type of this expression (using V and W as instances of δ) is then:

$$\Pi_{\text{sym}} V (\Pi_{\text{sym}} W (\text{Int} \rightarrow \text{PermState}_{\{\{X,Y\}\}}^{\text{rd}V \odot \text{rd}W \odot \text{wr}W} \text{Unit}))$$

Here $\text{rd}V$, $\text{rd}W$ and $\text{wr}W$ represent constants in the grading algebra associated with the read and write operations for a given symbol, and formally rd and wr have grade $\text{Symbol} \rightarrow \text{Effect}$ (so $\text{rd } W$ is application rather than a composite token, just

like `get X` at Value kind earlier). Notice the ‘ $\odot$ ’ operator in the grade – this is the *sequencing* operator of the grading algebras. Symbol-polymorphic terms are key for programming with graded monad builders.

Another graded monad builder is `Exception`, which represents computations that might raise exceptions. The grade of graded monadic types derived from `Exception` represents the set of exceptions possibly raised and not handled (similar to Java checked exceptions). Again writing E as an instance of variable symbol δ , the two main function we use are `raise` of type $\Pi_{\text{sym}} E (\text{Exception}_{\Sigma}^{\{E\}} A)$ and `handle` of type $\text{Exception}_{\Sigma}^F A \rightarrow \Pi_{\text{sym}} E (\text{Exception}_{\Sigma}^G \rightarrow \text{Exception}_{\Sigma}^{(F \setminus \{E\}) \cup G} A)$ for any type A and all grades F and G , with Σ as the underlying set of symbols for the graded monad. The former takes a single argument that raises the exception with the given name, whereas `handle` takes as input a computation M , a symbol representing E an exception, and another computation which should be executed if M raises and does not handle E . For example, we would write the following code to represent a ‘safe’ division function that raises an exception if the denominator is zero:

```
fn x  $\Rightarrow$  fn y  $\Rightarrow$  alt y = 0 then (return (x div y)) else (raise DivZero).
```

Assuming the exception set in this scope is $\Sigma = \{\text{DivZero}, \text{OutOfBounds}, \text{BadInput}\}$, the type of this expression is $\text{Int} \rightarrow \text{Int} \rightarrow \text{Exception}_{\Sigma}^{\{\text{DivZero}\}} \text{Int}$.

While division by zero is often a ubiquitous exception used throughout a program, we suppose the `BadInput` exception only matters in one part of the codebase and the rest does not need to consider it. It seems sensible to consider `BadInput` an exception that only exists within a lexical scope.¹ We therefore allow the programmer to define symbols that exist only within a certain lexical scope using `new X within M` syntax. A locally defined exception must stay local, never escaping the scope within which it is defined, as otherwise it cannot be handled.² Thus we can write code like the following:

```
let f = fn x  $\Rightarrow$  (new BadInput within extract (handle (alt isOK x then return x
                                                         else raise BadInput)
                                                         BadInput
                                                         (return 0))) in ...
```

Here function `extract` unwraps the result of an expression guaranteed not to raise any exceptions, meaning that its type is $\text{Exception}_{\Sigma}^{\{\}} A \rightarrow A$. This part of the program type-checks because the expression on the right-hand side of $f = (\dots)$ does not have a graded monadic type. This condition is imposed to reject expressions that might raise an exception represented by a symbol E and not handle it, thereby extruding E from its scope. Type-checking would reject replacing `extract` with the identity function.

These examples illustrate two simple analyses and how symbols are used within a programming language. The main concepts of symbol abstraction via `sym  $\delta \Rightarrow M$` syntax, the existence of named parameterised constants for grading algebras (`rd  $\Psi$`) and the introduction of lexically scoped symbols via `new X within M` are all shown. These are the key features of ML_G . Finally, analyses from Chapter 3, in particular live

¹In Java this can be done using a private class extending `Exception` that is only used in a private method of some class.

²Java checked exceptions require the name of any checked exception to remain in scope (otherwise the method type signature cannot be written).

variable analysis and deadlock-free locking, are also examples of graded monads that can be seen as instantiations of graded monad builders for a given set of symbols.

6.1.2 Definitions

Definition 6.1. Given a *symbol universe*, a finite set W of symbols, a *graded monad builder* $\mathcal{B}$ is a family $\mathcal{B}_S$ of graded monads, one for each $S \subseteq W$.

Informally, we can think of a graded monad builder as a mapping that produces a graded monad given a set of symbols. This means that the resulting grading algebra and the resulting operator are parameterised by the underlying symbol set – these are `PermState`, `Exception`, `LiveState` and `Par` in our examples. Note that we impose no relationship on $\mathcal{B}_S$ and $\mathcal{B}_{S'}$, though future work might investigate the idea of subtyping when $S \subseteq S'$.

6.2 System G: polymorphic symbols and grades

We now turn to System G, the first of two languages explored in this chapter. The approach System G takes when it comes to programming with graded monad builders is similar to System F: all types, grades, symbols and symbol sets have to be explicitly stated in order to achieve polymorphism. The aim of this language is to provide a way to define a formal semantics for graded monad builders.

We turn to the syntax and the type system of System G and then to its denotational semantics. Finally, we define an equational theory of System G, giving us a way of rewriting terms in a way that preserves semantics.

6.2.1 Syntax and types

System G is a six-kinded calculus: we have value kinds, types, grades, symbols, *symbol sets* and *graded monad builders*. The addition of symbol sets might be surprising at first – but there is a very good reason why these are an important separate kind. In the spirit of making everything explicit, the underlying algebra is also always specified whenever a graded monadic computation should be performed – and the algebra is generated by the underlying symbol set.

Syntactic categories

Syntactically, System G has six categories (Fig. 6.1). We use M and N as metavariables for value terms, A and B for types, F and G for grades, Ψ for symbol terms, S for symbol sets and T for graded monad builders. Let us start with symbols: a symbol Ψ is either a symbol variable (δ) or a symbol constant introduced in a particular scope (X). Next, a set of symbols S is either a symbol set-variable (ξ) or a symbol-set constant $\{X_1, \dots, X_n\}$.

For a particular graded monad builder T and a symbol set S , we are interested in both the underlying grading algebra of T instantiated with symbol set S and in the graded monadic type we get when T is instantiated with S , some grade F and type A . We use different notation for these two roles of T . We write $[T]S$ to refer, syntactically, to the grading algebra corresponding to the graded monad resulting

from instantiating T with S . We use the notation $T^\diamond S$ to refer to a graded monad type constructor, which is then applied to an effect and a type ($T^\diamond S F A$). Outside of the syntax of the language, we also encounter $T^\circ S$ in the semantics to refer to the grading algebra tuple (with carrier set, sequencing, equations...) that $[T]S$ corresponds to – we explain this in more detail in Section 6.2.2.

Grade-kinded terms (ranged over³ by F, G) represent either elements of a grading algebra, symbols or grading algebra operators. A grade can be the identity element of the algebra (referred to as 1), a grade variable (r, s), a product/sequencing of two grades (meaning the application of the monoid multiplication operator), or a nullary grade-level operator. Here grade-level operators with multiple parameters can be seen as ‘grade-kinded curried functions’ which can be applied to both symbols and grades. Hence F could also be an application of a function at the grade-term level. This matter is revisited when we define the judgements used by the type system of System G (described in section 6.2.1 and in Figure 6.3). Even though we defined computational grading algebras as join-semilattices, there is no join operator used in System G since we omit conditional branching.

Value-level terms (M, N) are firstly what we recognise expect from lambda calculus: they can be variables (x), λ -abstractions ($\lambda x:A (M)$), value-level applications ($M N$). The polymorphism of System F is present in System G, as abstractions and applications are introduced for symbols, symbol sets and for grades – but not for types (for the sake of making the denotational semantics simpler). In writing we use the same notation $M \Psi, M S$ and $M F$ for applications, and it is the syntactic element being applied to M that disambiguates what kind of application we are working with. When it comes to abstractions, we write: $\lambda_{\text{sym}} \delta:S (M)$ to abstract over a symbol drawn from a set of symbols S ; $\lambda_{\text{ss}} \xi (M)$ to abstract over a symbol set; $\Lambda_{\text{eff}} r:[T]S (M)$ to abstract over a grade drawn from the grading algebra $[T]S$. The reason why Λ is used over λ for grade abstraction is because a language in which all grades are inferred would not feature this construct (similar to Hindley–Milner polymorphism in relation to System F). Finally, syntax $\nu X (M)$ introduces new symbols.

Types (A, B) have six forms; base types and function types are standard. Graded monadic types are also present, written as $T^\diamond S F A$, where T is a graded monad builder, S is a symbol set with which T gets instantiated to give a graded monad, F is a grade and A is a type (following our intuition that the kind of graded monad builders is $\text{SymbolSet} \rightarrow \text{Grade} \rightarrow \text{Type} \rightarrow \text{Type}$), representing types $T_S^F A$ seen previously. In addition, each of the abstractions of System G gives a new polymorphic type or ‘function type’. Aside from $A_1 \rightarrow A_2$, we have $\forall_{\text{eff}} r:[T]S (A)$ types for grade polymorphism. The $[T]S$ in the grade abstraction is used to state the set over which r ranges – that being the carrier set of the grading algebra $[T]S$. Symbol abstraction has its own corresponding type $\Pi_{\text{sym}} \delta:S (A)$, where S specifies the set from which δ is drawn. Finally, symbol set abstractions have type $\Pi_{\text{ss}} \xi (A)$. We omit type-level polymorphism as we want to focus on grade, symbol and symbol set variables. While polymorphic types are easy to add, type polymorphism as seen in System F complicates the denotational semantics; one approach is that of *hyperdoctrines* [Pit87].⁴

³Previously we used F to range over grades; by abuse of notation it now ranges over grade-kinded terms which include grades (now ranged over by r and some members of O) as a special case.

⁴Hyperdoctrines originate in Lawvere’s investigation of adjoint functors in formal logic and proof theory. [Law69] They are commonly used to represent logical systems with context manipulation and quantification over propositions/types. Thus, in addition to a ccc $\mathbb{C}$ whose objects are types and whose

value terms	$M, N ::=$	$x \mid c \mid \nu X (M) \mid \lambda x:A (M) \mid M N$ $\mid \lambda_{\text{sym}} \delta:S (M) \mid M \Psi$ $\mid \lambda_{\text{ss}} \xi (M) \mid M S$ $\mid \Lambda_{\text{eff}} r:[T] S (M) \mid M F$ $\mid M \gg_{T \blacktriangleright S} N \mid \text{ret}_{T \blacktriangleright S} M \mid \text{sub}_{T \blacktriangleright S} F G M$
value-level types	$A, B ::=$	$\text{BaseType} \mid A_1 \rightarrow A_2 \mid T \blacktriangleright S F A$ $\mid \forall_{\text{eff}} r:[T] S (A)$ $\mid \Pi_{\text{sym}} \delta:S (A) \mid \Pi_{\text{ss}} \xi (A)$
symbols	$\Psi ::=$	$\delta \mid X$
symbol sets	$S ::=$	$\xi \mid \{\{X_1, \dots, X_n\}\}$
grade terms	$F, G ::=$	$\mathbf{1} \mid r \mid O \mid F \odot G \mid F G \mid F \Psi$
grade-level types	$\varphi ::=$	$\text{Eff} \mid \text{Eff} \Rightarrow \varphi \mid \text{Sym} \Rightarrow \varphi$
graded monad builders	$T ::=$	$\text{Exception}, \text{StLive}, \text{StPerm}, \text{etc. see Fig. D.1}$

Figure 6.1. The grammar of System G terms: values, value types, symbols, symbol sets, grades, grade types, and graded-monad builders. See Appendix for details about graded monad builders.

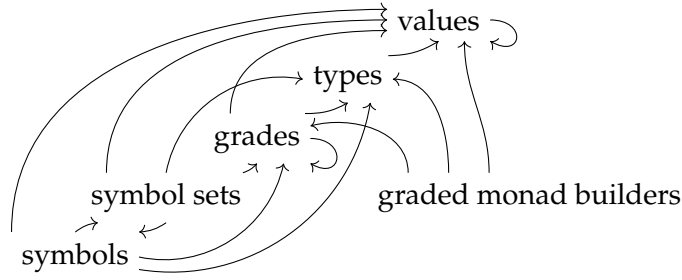


Figure 6.2. Relationship between System G kinds. An arrow $k_1 \rightarrow k_2$ means that elements of kind k_2 are built out of elements of kind k_1 . Note the lack of values $\rightarrow$ types arrow, meaning that there are no dependent types, and the lack of an arrow types $\rightarrow$ types, meaning there is no type polymorphism.

The relationship between the six kinds can be visualised as in Figure 6.2. Values are built out of abstractions over other values, over types, symbols, symbol sets and grades. Types can depend on and contain names of grades, symbols and symbol sets. Grade terms can contain applications of grade-level operators to symbols and they may contain grade variables drawn from a grading algebra, which is generated by the underlying set of symbols.

Kind and type judgements

The type system of System G naturally extends the type system of System F. One part of formally specifying the type system of System F is having a separate context

morphisms are terms (as in the categorical semantics of STLC), hyperdoctrines additionally assume for each type A a category PA representing the type's attributes and deductions related to this type, as well as ways of representing context extension, substitution and quantification. Hyperdoctrines model first-order intuitionistic logics with equality. [See83]

of type variables and a separate typing context. System G takes the logical next step: the context comprises of grade, symbol and symbol set variables, as well as an environment of introduced symbols via $\nu X (M)$. This *inference context*, which we refer to as $\mathcal{J}$, has five components:

- Σ the set of symbol constants, introduced via $\nu X (M)$;
- $\mathcal{S}$ symbol set variables, bound in abstractions $\lambda_{ss} \xi (M)$;
- Δ the set of symbol variables, introduced through $\lambda_{sym} \delta : S (M)$;
- Φ grade variables, bound in abstractions $\Lambda_{eff} r : [T]S (M)$ (meaning r represents an effect drawn from grading algebra $[T]S$);
- Γ the typing context of value-level variables, introduced via $\lambda x : A (M)$.

Given a typing context $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \mid \Gamma$ we can subset its components using notation such as $\mathcal{J}_{\Sigma\Gamma}$ to select only its Σ and Γ fields. Similarly we write $\mathcal{J}\{f \mapsto \phi\}$ to refer to a context that is identical to $\mathcal{J}$ except for the field f , which is substituted with ϕ : for example, $\mathcal{J}\{\Sigma \mapsto X, \mathcal{J}_{\Sigma}\}$ is the context in which the Σ field of $\mathcal{J}$ is extended with symbol X . Finally, we also write $\mathcal{J}_{f_1, f_2}$ to extract the sub-context with only fields f_1 and f_2 .

The type system has various type-like judgements, first:

Well-typed term The central, and traditional, $\mathcal{J} \vdash_{val} M : A$. This relies on several auxiliary judgements, mostly regarding well-formedness of non-value constructs in the language.

Valid symbol sets A judgement of the form $\Sigma \mid \mathcal{S} \vdash_{ss} S$ represents the fact that S is a valid symbol set if the set of introduced symbol constants in a particular lexical scope is Σ , and $\mathcal{S}$ is the set of symbol set variables.

Valid symbol A judgement of the form $\Sigma \mid \mathcal{S} \mid \Delta \vdash_{sym} \Psi \in S$ represents the fact that Ψ is a valid symbol and it belongs to set S , given the set of introduced symbol constants Σ and the set of free symbol variables Δ in a particular lexical scope.

Valid type A judgement of the form $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{type} A$ represents the fact that A is a valid type in a given context (all components other than Γ are necessary).

Valid grade A judgement of the form $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{gr} F \triangleleft [T]S \& \varphi$ represents the fact that, given a context $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi$, a grade-level term F is interpretable in the grading algebra $[T]S$, and it has the *grade-level type* φ .

Subeffecting We write $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{sub} F \sqsubseteq_{[T]S} G$ for subeffecting judgements, that is, saying that F and G are both effects drawn from the grading algebra $[T]S$ in context $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi$, and furthermore F is the subeffect of G . These judgements are separately defined for each T and S and must result in this relation between $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi, [T]S, F$ and G being decidable. Hence we do not define them in this chapter.⁵

⁵Soundness of these judgements is important, but completeness is not: there can be some grade-kinded terms F and G such that F is a subeffect of G in a given context, but this fact is not derivable for the rules associated with $[T]S$.

The ‘valid grade’ judgements are the only one that are not straightforward and syntactic, so they require additional explanation. As mentioned when introducing the syntax, grade-level terms not only include grades, but also grade-level functions. This motivates *grade-level typing*, which ensures that grade-kinded terms are used properly. The φ in $\Sigma \mid \Delta \mid \Phi \vdash_{\text{gr}} F \triangleleft [T]S$ & φ is a grade-level type, whose syntax is given in Figure 6.1.

The above judgements are defined inductively (Figures 6.3 and 6.4) using axioms and rules. None of these rules is particularly interesting except for (EFF-CONST), which states the grade-level type of a grade-level constant O : the grade-level type is specified by the grading algebra(s) associated with graded monad builder T .

With these judgements defined, we can turn to a short simple example: the value-kinded term

$$\lambda_{\text{ss}} \xi (\lambda_{\text{sym}} \delta_1 : \xi (\lambda_{\text{sym}} \delta_2 : \xi (\text{getPerm } \delta_1 \gg_{\text{StPerm} \blacklozenge \xi} \text{putPerm } \delta_2)))$$

has type:

$$\Pi_{\text{ss}} \xi (\Pi_{\text{sym}} \delta_1 : \xi (\Pi_{\text{sym}} \delta_2 : \xi (\text{StPerm} \blacklozenge \xi (\text{rdPerm } \delta_1 \odot \text{wrPerm } \delta_2) \text{Unit})))$$

6.2.2 Denotational semantics

For the simply typed λ calculus, the semantics of a term depends on its type derivation. We thus have $\llbracket \Gamma \vdash M : A \rrbracket$ being a member of domain $\llbracket \Gamma \rrbracket \rightarrow \llbracket A \rrbracket$, noting the overloading of $\llbracket \cdot \rrbracket$ with three related meanings. We use ρ to range over $\llbracket \Gamma \rrbracket$ so we might write a typical case as:

$$\llbracket \Gamma \vdash x : A \rrbracket \rho = \rho(x)$$

For System G terms the situation is parallel, except that our context $\mathcal{J}$ for such judgements has five components: $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \mid \Gamma$. Similarly the analogue of ρ , written $\mathcal{R}$, has five components which we write using record-style field-selection: $\mathcal{R}_\Sigma, \mathcal{R}_\mathcal{S}, \mathcal{R}_\Delta, \mathcal{R}_\Phi$ and $\mathcal{R}_\Gamma$. The last of these plays the role of ρ above. We can see $\mathcal{J}$ and $\mathcal{R}$ as generalised ‘type’ and ‘value’ environments of five components. Moreover, System G types themselves depend on an environment, hence the denotation of a term belongs to the domain as shown:

$$\llbracket \mathcal{J} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} \mathcal{R} \in \llbracket \mathcal{J}_{\Sigma, \mathcal{S}, \Delta, \Phi} \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R}_{\Sigma, \mathcal{S}, \Delta, \Phi}$$

Note that the $\vdash_{\text{type}}$ judgement only uses a four-element subset of components of $\mathcal{J}$; we use the same notation to select component-subsets of $\mathcal{R}$.

The full denotational semantics of System G is based on defining the semantics of its various constructs, particularly its six kinds of terms and its five context components. The six semantic functions $\llbracket \cdot \rrbracket_{\text{val}}, \llbracket \cdot \rrbracket_{\text{type}}, \llbracket \cdot \rrbracket_{\text{gr}}, \llbracket \cdot \rrbracket_{\text{ss}}, \llbracket \cdot \rrbracket_{\text{sym}}, \llbracket \cdot \rrbracket^{\text{gmb}}$ respectively give meaning to (entailments of) terms of kinds value M , type A , effect F , symbol set S , symbol Ψ and monad builder T . Similarly, the five context components of $\mathcal{J}$, being $\Sigma, \mathcal{S}, \Delta, \Phi$ and Γ , are given meaning by semantic functions $\llbracket \cdot \rrbracket^{\text{symcenv}}, \llbracket \cdot \rrbracket^{\text{ssenv}}, \llbracket \cdot \rrbracket^{\text{sym2ss}}, \llbracket \cdot \rrbracket^{\text{gr2alg}}, \llbracket \cdot \rrbracket^{\text{val2type}}$. Finally, (syntactic) grading algebras associated with a graded monad builder and a symbol set also have their denotations as computational grading algebras, given by the semantic function $\llbracket \cdot \rrbracket^{\text{alg}}$.

The semantics of terms in context is, in all six cases, a function that maps records $\mathcal{R}$ representing the ‘run-time values’ in a context (in a sense, it is a run-time analogue of $\mathcal{J}$) to the semantics of the term.

The definition of the denotational semantics is deferred to Appendix D.2.

$$\begin{array}{c}
\text{(SS-VAR)} \frac{\xi \in \mathcal{S}}{\Sigma \mid \mathcal{S} \vdash_{\text{ss}} \xi} \quad \text{(SS-CONST)} \frac{X_1, \dots, X_n \in \Sigma}{\Sigma \mid \mathcal{S} \vdash_{\text{ss}} \{\{X_1, \dots, X_n\}\}} \\
\\
\text{(SYM-}\Xi\text{-VAR)} \frac{\Delta(\delta) = S}{\Sigma \mid \mathcal{S} \mid \Delta \vdash_{\text{sym}} \delta \in S} \\
\text{(SYM-}\Xi\text{-CONST)} \frac{\Sigma \mid \mathcal{S} \vdash_{\text{ss}} \{\{X_1, \dots, X_n\}\} \quad 1 \leq i \leq n}{\Sigma \mid \mathcal{S} \mid \Delta \vdash_{\text{sym}} X_i \in \{\{X_1, \dots, X_n\}\}} \\
\\
\text{(ALG-VALID)} \frac{\Sigma \mid \mathcal{S} \vdash_{\text{ss}} S}{\Sigma \mid \mathcal{S} \vdash_{\text{alg}} T^\circ S} \\
\\
\text{(EFF-VAR)} \frac{\Phi(r) = [T]S}{\mathcal{J} \vdash_{\text{gr}} r \triangleleft [T]S \ \& \ \text{Eff}} \quad \text{(EFF-IDENT)} \frac{}{\mathcal{J} \vdash_{\text{gr}} \mathbf{1} \triangleleft [T]S \ \& \ \text{Eff}} \\
\text{(EFF-MULT)} \frac{\mathcal{J} \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Eff} \quad \mathcal{J} \vdash_{\text{gr}} G \triangleleft [T]S \ \& \ \text{Eff}}{\mathcal{J} \vdash_{\text{gr}} F \odot G \triangleleft [T]S \ \& \ \text{Eff}} \\
\text{(EFF-CONST)} \frac{O \text{ is a constant with grade-level type } \varphi \text{ associated with } [T]S}{\mathcal{J} \vdash_{\text{gr}} O \triangleleft [T]S \ \& \ \varphi} \\
\text{(EFF-EFFAPP)} \frac{\mathcal{J} \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Eff} \Rightarrow \varphi \quad \mathcal{J} \vdash_{\text{gr}} G \triangleleft [T]S \ \& \ \text{Eff}}{\mathcal{J} \vdash_{\text{gr}} F G \triangleleft [T]S \ \& \ \varphi} \\
\text{(EFF-EFFSYM)} \frac{\mathcal{J} \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Sym} \Rightarrow \varphi \quad \Sigma \mid \mathcal{S} \mid \Delta \vdash_{\text{sym}} \Psi \in S}{\mathcal{J} \vdash_{\text{gr}} F \Psi \triangleleft [T]S \ \& \ \varphi} \\
\\
\text{(TYPE-BASE)} \frac{}{\mathcal{J} \vdash_{\text{type}} \text{BaseTypes}} \quad \text{(TYPE-FN)} \frac{\mathcal{J} \vdash_{\text{type}} A_1 \quad \mathcal{J} \vdash_{\text{type}} A_2}{\mathcal{J} \vdash_{\text{type}} A_1 \rightarrow A_2} \\
\text{(TYPE-MONAD)} \frac{\mathcal{J} \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Eff} \quad \mathcal{J} \vdash_{\text{type}} A}{\mathcal{J} \vdash_{\text{type}} T^\blacklozenge S F A} \\
\text{(TYPE-ALLEFF)} \frac{\Sigma \mid \mathcal{S} \mid \Delta \mid r:[T]S, \Phi \vdash_{\text{type}} A}{\mathcal{J} \vdash_{\text{type}} \forall_{\text{eff } r:[T]S} (A)} \\
\text{(TYPE-PISYM)} \frac{\Sigma \mid \mathcal{S} \mid \delta:S, \Delta \mid \Phi \vdash_{\text{type}} A}{\mathcal{J} \vdash_{\text{type}} \Pi_{\text{sym}} \delta:S (A)} \quad \text{(TYPE-PISSET)} \frac{\Sigma \mid \xi, \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} A}{\mathcal{J} \vdash_{\text{type}} \Pi_{\text{ss}} \xi (A)}
\end{array}$$

Figure 6.3. Symbol, Grade and Type entailment judgements. Here $\mathcal{J}$ abbreviates $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi$.

(CONST) $\frac{}{\mathcal{J} \vdash_{\text{val}} c : \mathcal{T}_c}$
(VAR) $\frac{\Gamma(x) = A}{\mathcal{J} \vdash_{\text{val}} x : A}$ (NEW) $\frac{X, \Sigma \mathcal{S} \Delta \Phi \Gamma \vdash_{\text{val}} M : A}{\mathcal{J} \vdash_{\text{val}} \nu X (M) : A}$ provided $X \notin \text{sym}(A)$
(VAL-ABS) $\frac{\Sigma \mathcal{S} \Delta \Phi x : A_1, \Gamma \vdash_{\text{val}} M : A_2}{\mathcal{J} \vdash_{\text{val}} \lambda x : A_1 (M) : A_1 \rightarrow A_2}$
(VAL-APP) $\frac{\mathcal{J} \vdash_{\text{val}} M : A_1 \rightarrow A_2 \quad \mathcal{J} \vdash_{\text{val}} N : A_1}{\mathcal{J} \vdash_{\text{val}} M N : A_2}$
(VAL-SYM-ABS) $\frac{\Sigma \mathcal{S} \delta, \Delta \Phi \Gamma \vdash_{\text{val}} M : A}{\mathcal{J} \vdash_{\text{val}} \lambda_{\text{sym}} \delta : S (M) : \Pi_{\text{sym}} \delta : S (A)}$
(VAL-SYM-APP) $\frac{\mathcal{J} \vdash_{\text{val}} M : \Pi_{\text{sym}} \delta : S (A) \quad \Sigma \mathcal{S} \Delta \vdash_{\text{sym}} \Psi \in S}{\mathcal{J} \vdash_{\text{val}} M \Psi : A\{\Psi/\delta\}}$
(VAL-SS-ABS) $\frac{\Sigma \xi, \mathcal{S} \Delta \Phi \Gamma \vdash_{\text{val}} M : A}{\mathcal{J} \vdash_{\text{val}} \lambda_{\text{ss}} \xi (M) : \Pi_{\text{ss}} \xi (A)}$
(VAL-SS-APP) $\frac{\mathcal{J} \vdash_{\text{val}} M : \Pi_{\text{ss}} \xi (A) \quad \Sigma \mathcal{S} \vdash_{\text{ss}} S}{\mathcal{J} \vdash_{\text{val}} M S : A\{S/\xi\}}$
(VAL-EFF-ABS) $\frac{\Sigma \mathcal{S} \Delta r : [T]S, \Phi \Gamma \vdash_{\text{val}} M : A}{\mathcal{J} \vdash_{\text{val}} \Lambda_{\text{eff}} r : [T]S (M) : \forall_{\text{eff}} r : [T]S (A)}$
(VAL-EFF-APP) $\frac{\mathcal{J} \vdash_{\text{val}} M : \forall_{\text{eff}} r : [T]S (A) \quad \Sigma \mathcal{S} \Delta \Phi \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Eff}}{\mathcal{J} \vdash_{\text{val}} M F : A\{F/r\}}$
(BIND) $\frac{\mathcal{J} \vdash_{\text{val}} M : T^\blacklozenge S F A_1 \quad \mathcal{J} \vdash_{\text{val}} N : A_1 \rightarrow T^\blacklozenge S G A_2}{\mathcal{J} \vdash_{\text{val}} M \gg_{T^\blacklozenge S} N : T^\blacklozenge S (F \odot G) A_2}$
(RET) $\frac{\mathcal{J} \vdash_{\text{val}} M : A \quad \Sigma \mathcal{S} \vdash_{\text{ss}} S}{\mathcal{J} \vdash_{\text{val}} \text{ret}_{T^\blacklozenge S} M : T^\blacklozenge S \mathbf{1} A}$ (SUB) $\frac{\mathcal{J} \vdash_{\text{val}} M : T^\blacklozenge S F A \quad \mathcal{J} \vdash_{\text{sub}} F \sqsubseteq_{[T]S} G}{\mathcal{J} \vdash_{\text{val}} \text{sub}_{T^\blacklozenge S} F G M : T^\blacklozenge S G A}$

Figure 6.4. Typing rules for value terms. Here $\mathcal{J}$ abbreviates $\Sigma | \mathcal{S} | \Delta | \Phi | \Gamma$. We write $\text{sym}(A)$ for the set of symbol constants occurring in type A . $\mathcal{T}_c$ is the base type of which c is a member.

6.2.3 Equational theory

An important component of System G is its equational theory, essentially giving us a set of equations representing semantically equivalent terms. A theory of equivalent terms and rewriting is essential for making System G a calculus. In this section we define the equational theory of System G and show that the rewriting rules are sound with respect to the denotational semantics.

We now define the equivalence relation $\mathcal{J} \vdash M \equiv M' : A$ on terms M and M' , where $\mathcal{J}$ is a context and A is the type of M and M' . First we note it is reflexive in M : $\mathcal{J} \vdash M \equiv M : A$ for all M such that $\mathcal{J} \vdash M : A$. Next we consider contextual equivalence, which for terms is given by:

$$\mathcal{J} \vdash C[M] \equiv C[M'] : A \quad \text{for } \mathcal{J} \vdash M \equiv M' : B \text{ and } C \text{ a (term) context of type } A$$

and similarly for contexts of other kinds. The other equations are all variants of $\beta\eta$ equivalence for all abstraction terms in System G. We use $fv(M)$, $fev(M)$, $fsv(M)$ and $fssv(M)$ to respectively refer to free variables, free grade variables, free symbol variables and free symbol set variables occurring in M .

$$\begin{array}{llll} \mathcal{J} \vdash (\lambda x:A (M)) N & \equiv & M\{N/x\} & : B \\ \mathcal{J} \vdash (\Lambda_{\text{eff}} r:[T] S (M)) F & \equiv & M\{F/r\} & : B \\ \mathcal{J} \vdash (\lambda_{\text{sym}} \delta:S (M)) \Psi & \equiv & M\{\Psi/\delta\} & : B \\ \mathcal{J} \vdash (\lambda_{\text{ss}} \xi (M)) S & \equiv & M\{S/\xi\} & : B \\ \mathcal{J} \vdash \lambda x:A (M x) & \equiv & M & : B \rightarrow A \quad \text{for } x \notin fv(M) \\ \mathcal{J} \vdash \Lambda_{\text{eff}} r:[T] S (M r) & \equiv & M & : \forall_{\text{eff}} r:[T] S (A) \quad \text{for } r \notin fev(M) \\ \mathcal{J} \vdash \lambda_{\text{sym}} \delta:S (M \delta) & \equiv & M & : \Pi_{\text{sym}} \delta:S (A) \quad \text{for } \delta \notin fsv(M) \\ \mathcal{J} \vdash \lambda_{\text{ss}} \xi (M \xi) & \equiv & M & : \Pi_{\text{ss}} \xi (A) \quad \text{for } \xi \notin fssv(M) \end{array}$$

All of these equivalences produce semantically equivalent terms. This is stated formally by the following *soundness* theorem:

Theorem 6.1. Two equivalent terms have the same denotation: if $\mathcal{J} \vdash M \equiv M' : A$, then:

$$\llbracket \mathcal{J} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} = \llbracket \mathcal{J} \vdash_{\text{val}} M' : A \rrbracket_{\text{val}}$$

The significance of this result is that $\beta\eta$ conversions, and other equivalences based on $\equiv$, preserve semantics – as expected.

6.3 ML_G: inferring polymorphic grades

Thus far in the thesis, our focus was mainly on the Source language. In this section, we introduce ML_G, a small language whose purpose is to show that graded monad builders and decidable type inference can work together.

We first turn to the syntax of ML_G, which for the most part resembles that of Haskell and the ML language family. We follow this up by introducing and explaining the type system of ML_G, which is polymorphic in the style of Hindley-Damas-Milner [Hin69, DM82], and features a separate kind of grades. The type system admits type inference, which we address in Section 6.3.2.

The semantics of ML_G is defined via elaboration into System G, similar to what was done in Chapter 4. A point to keep in mind is that System G does not have type polymorphism, so restrictions are placed on ML_G terms that are being translated.

6.3.1 Syntax and types

Syntax of ML_G

The main syntactic elements of ML_G are *value terms* (M, N) and *symbols* (Ψ). We also have *types* (A, B) and *grades* (F, G), which we define in Section 6.3.1.

Symbol terms $\Psi ::= \delta \mid X$ are as in System G: a symbol is either a *symbol variable* (δ) or a *constant symbol* (X , ranges over $X, Y, \dots$). Symbol variables are introduced via function-style abstractions $\text{sym } \delta \Rightarrow M$ (similarly to variables at the value level being introduced via $\text{fn } x \Rightarrow M$). Constant symbols are introduced with the $\text{new } X \text{ within } M$ construct (which we turn to later in this section). The symbol sets $S ::= \xi \mid \{X_1, \dots, X_n\}$ of System G are omitted from ML_G: instead a graded monad builder type is implicitly applied to the symbol set of symbol constants in scope due to $\text{new } X \text{ within } M$ constructs (see Section 6.3.1 below). While values and symbols use explicit lambda-style abstraction, to keep the language simple we use implicit ML-style (rather than explicit System-F-style) polymorphism for type and grade variables.

Value terms M are given by the following grammar:

$$\begin{aligned} M ::= & c \mid x \mid \text{fn } x \Rightarrow M \mid M_1 M_2 \mid \text{let } x = M_1 \text{ in } M_2 \\ & \mid \text{return } M \mid \text{bind } x \Leftarrow M_1 \text{ with } M_2 \mid \text{new } X \text{ within } M \\ & \mid \text{sym } \delta \Rightarrow M \mid M \Psi \end{aligned}$$

The first line is standard, constants (c), variables (x), function abstraction and application, as well as let-expressions. The second line starts with graded monadic constructs: $\text{return } M$ corresponds to wrapping an expression inside a graded monad like with return (with the corresponding pure/identity grading), whereas $\text{bind } x \Leftarrow M_1 \text{ with } M_2$ is just the monadic bind in which x appears freely in M_2 (in other settings this is written as $M_1 \gg= \lambda x. M_2$). The remaining forms involve both value expressions and grades. The construct $\text{new } X \text{ within } M$ introduces/binds a ‘new symbol’ X to the grading algebra in the current scope, which allows M to use X or have a type that depends on X . Finally, we have abstraction of symbols at the value level, as well as applications of expressions to symbols. Note that the syntax above provides no support for looping or recursion – this was discussed in Chapter 5, the reason is the issue of polymorphic recursion (support for monomorphic recursion requires explicit type annotation, which we are avoiding entirely in ML_G).

We do not add products, sums, conditional statements and other constructs useful in practice – mainly because adding them to the language is straightforward.

Type system

We distinguish between *types* (A, B) and *grades* (F, G).

ML-like languages have various type-inference issues concerning side-effecting operations and scope extrusion (of types and type variables); these are resolved by language restrictions, specifically the “value restriction” and the bans on top-level polymorphic exceptions and exporting values referring to local type definitions. While ML_G avoids these problems by having explicit monads and no locally scoped type definitions we do have scoped grade variables which raise similar issues. We enforce a ban on grade scope extrusion by distinguishing between two categories of types: general types A and a sub-category of pure types τ which contain no graded

monadic types. These ensure that symbols constants cannot ‘leak’ outside the scope which introduces them (scope extrusion is banned).

The language also has two levels of parametric polymorphism: one for grades and another for types. This means there are two different kind of $\forall$ -bound variables at the type level (grade variables and type variables). This requires defining the notion of *type schemes*, which extend the ones seen in Hindley-Milner-Damas polymorphic type systems.

The types A and τ and type schemes σ in ML_G are given by the following grammar:

$$\begin{aligned}\sigma &::= \forall_{\text{eff}} r. \sigma \mid \forall_{\text{type}} \alpha. \sigma \mid A \\ A, B &::= \text{BaseTypes} \mid \alpha \mid A_1 \rightarrow A_2 \mid T^F A \mid \Pi_{\text{sym}} \delta (A) \\ \tau &::= \text{BaseTypes} \mid \alpha \mid \tau_1 \rightarrow \tau_2\end{aligned}$$

The base types, type variables and function types are as usual. The graded monadic type $T^F A$ is used here with no explicit specification of the symbol set – graded monad builders use the set of constant symbols in scope as the symbol set S previously written as $T^\bullet S F A$, but here we assume that the symbol set that the symbol is drawn from is Σ (set of symbols in scope introduced via `new X within ...`). Finally, the type $\Pi_{\text{sym}} \delta (A)$ is similar to the function type, except that it expects a symbol as an argument rather than a value.

We commonly write type schemes using sets of grade and type variables: $\sigma = (\forall_{\text{eff}} R)(\forall_{\text{type}} V)A$. When $R = \{\}$, we write the type scheme as $\sigma = (\forall_{\text{type}} V)A$ for convenience. Similarly we write $\sigma = (\forall_{\text{eff}} R)A$ when $V = \{\}$, and $\sigma = A$ when $R = V = \{\}$.

Notice that there is no $\forall$ quantifier for symbols: new symbols are either introduced with the `new X within M` syntax and existing symbols can be given as arguments to expressions of type $\Pi_{\text{sym}} \delta (A)$.

A grade expression F is defined in a way similar to System G:

$$\begin{aligned}F, G &::= \mathbf{1} \mid r \mid F_1 \odot F_2 \\ &\quad \mid O \Psi_1 \dots \Psi_n F_1 \dots F_m \quad \text{for } O \in \mathcal{O} \text{ and } \text{arity}(O) = n + m\end{aligned}$$

Grade variables, similarly to type variables, are not explicitly introduced by the programmer and there is no possibility of abstracting over them – they are $\forall$ -bound. There are no explicit grade-level types in ML_G .

Definition 6.2. A type A *specialises* a type scheme σ , where

$$\sigma = (\forall_{\text{eff}} r_1, \dots, r_m)(\forall_{\text{type}} \alpha_1, \dots, \alpha_n)A'$$

(written $\sigma > A$) if A can be obtained from A' by simultaneously substituting some grades F_i and types τ_j for the grade variables r_i and type variables α_j (for $i = 1, \dots, m$ and $j = 1, \dots, n$):

$$A = A'[r_1 \mapsto F_1, \dots, r_m \mapsto F_m, \alpha_1 \mapsto A_1, \dots, \alpha_n \mapsto A_n]$$

[The notation $A'[r_1 \mapsto F_1, \dots, \alpha_m \mapsto A_m]$ represents the application of a substitution mapping r_1 to F_1 and so on to type A' .]

We furthermore introduce *type subsumption* judgements $\Sigma \mid \Delta \Vdash A <: B$ representing the fact that in a particular symbol context $\Sigma \mid \Delta$ type A is a subtype of B . The first

$$\begin{array}{c}
\text{(REFL-}<: \text{)} \frac{}{\Sigma \mid \Delta \Vdash A <: A} \\
\\
\text{(TRANS-}<: \text{)} \frac{\Sigma \mid \Delta \Vdash A_1 <: A_2 \quad \Sigma \mid \Delta \Vdash A_2 <: A_3}{\Sigma \mid \Delta \Vdash A_1 <: A_3} \\
\\
\text{(FUN-}<: \text{)} \frac{\Sigma \mid \Delta \Vdash A'_1 <: A_1 \quad \Sigma \mid \Delta \Vdash A_2 <: A'_2}{\Sigma \mid \Delta \Vdash A_1 \rightarrow A_2 <: A'_1 \rightarrow A'_2} \\
\\
\text{(GRADED-}<: \text{)} \frac{\Sigma \mid \Delta \Vdash A_1 <: A_2 \quad \Sigma \mid \Delta \Vdash_{\text{sub}} F \sqsubseteq G}{\Sigma \mid \Delta \Vdash T^F A_1 <: T^G A_2} \\
\\
\text{(DEP-}<: \text{)} \frac{\Sigma \mid \delta, \Delta \Vdash A <: A'}{\Sigma \mid \Delta \Vdash \Pi_{\text{sym}} \delta (A) <: \Pi_{\text{sym}} \delta (A')}
\end{array}$$

Figure 6.5. Rules defining the type subsumption judgements $\Sigma \mid \Delta \Vdash A_1 <: A_2$.

reason for introducing it is to simplify grades inside types in presence of operations – for example, type $T^{\{X\} \cup \{Y\}} A$ is the same as $T^{\{X, Y\}} A$ because $\{X\} \cup \{Y\} = \{X, Y\}$. The second is for the sake of expressing subeffecting as subtyping: if $F \sqsubseteq G$, then $T^F A <: T^G A$ for any type A . The full definition of the type subsumption relation is given in Figure 6.5. The ‘subeffecting leads to subtyping’ rule is (GRADED- $<:$) and it omits the symbol context stating the sets of symbols used by F and G . The (GRADED- $<:$) rule makes use of judgements $\Sigma \mid \Delta \Vdash_{\text{sub}} F \sqsubseteq G$. Derivability and decidability of these judgements, as in System G, depends on the grading algebra in question.

The definition of the type system makes use of typing judgements of the form $\Sigma \mid \Delta \mid \Gamma \Vdash M : A$, where Σ is the *symbol constant context*, Δ is the *symbol variables context*, Γ is the *variable context*. As usual Γ is a finite partial function from the set of value variables to type schemes. The symbol contexts Σ and Δ are the sets of all symbols which can freely appear in M and A ; the former contains the symbols introduced using the **new** X **within** M syntax, while the latter has those symbols introduced via **sym** $\delta \Rightarrow M$.

We additionally require symbol judgements of the form $\Sigma \mid \Delta \Vdash_{\text{sym}} \Psi$, which are defined in a straightforward way: $\Sigma \mid \Delta \Vdash_{\text{sym}} \Psi$ when $\Psi \in \Sigma \cup \Delta$.

These entailment show up in the typing rules – which can now finally be stated (Figure 6.6). Special attention should be paid to (SYM-NEW), where we use τ rather than A , with the aim to prevent graded monadic values leaking outside the scope, and the *prune*($\cdot$) function can be thought of as ‘projection’ of a typing environment onto one in which only τ ’s appear.

$$\begin{aligned}
\text{prune}(\diamond) &= \diamond \\
\text{prune}(\Gamma, x : A) &= \begin{cases} \text{prune}(\Gamma), x : A, & A \text{ is a } \tau \\ \text{prune}(\Gamma), & A \text{ is not a } \tau \end{cases}
\end{aligned}$$

This prevents us from writing functions and values which operate on graded monadic computations and then using them in a context in which a new symbol constant is introduced: **let** $m = \text{raise } X \text{ in new } Y \text{ within } \dots f \dots$. Another notable rule is (SUB) – it is introduced in order to impose that, if A_1 is a subtype of A_2 , then any expression of type A_1 also has type A_2 .

Requirements on instances All applications (separate instances of ML_G in which different computational effects are considered) of graded monad builders require us to use various additional functions, but they should not be completely arbitrary:

- new effectful computations and functions should have types that can be expressed in ML_G and not require additions to the type system;
- all grade-level operators should be members of the $\mathcal{O}$ set in the grading algebra.

In addition, we permit instances to add functions that take an argument of type $T^F A$ and return a function whose signature depends on the *number of symbol constants* in the symbol context Σ . This addition seems curious, but there is a justification for it. Monads (and graded monads) usually represent some sort of computation, with the aim to eventually extract its result. Getting this result requires a run function. For example, the **State** s monad deals with computations that involve a single cell. The `runState` function takes a monadic value wrapped in **State** and returns a function from the initial value of the mutable store and returns a pair of result and final value in the store, that is, its type is **State** $s \alpha \rightarrow s \rightarrow (\alpha \times s)$ where s is the type of the store and α is the type of the result. For a **TwoPermState** graded monad (with two integer states), the `runTwoPermState` function should ideally have type **TwoPermState** $\alpha \rightarrow \mathbb{Z} \rightarrow \mathbb{Z} \rightarrow (\alpha \times \mathbb{Z} \times \mathbb{Z})$. The reason we are looking at the number of symbol constants rather than all symbols is due to the fact that symbol abstraction does not modify the underlying graded monad, and thus the run signature should not change.

Scope matters The (SYM-NEW) rule differs from the others in that it uses a τ type in the premise and conclusion, rather than A , meaning that the type must not be comprised of any graded monadic types or abstractions over symbols. A graded monad builder T applied to symbol set $\{X\} \cup \Sigma$ differs from the one in symbol context Σ , we just use the same symbol to refer to them in ML_G . The former limitation was justified in Section 6.1, whereas the latter is not immediately obvious. Consider the following program dealing with stateful computation:

```
new X within
  let z =
    new Y within
      ... get X ...
    in ... z ...
```

Each `new X within` M introduces a new symbol, as well as a new lexically scoped graded monad (even though we call them both T). Note that context determines the graded monadic type given to expression `get X`; in the above the grading algebra consist of both X and Y , so the the type can involve both of these symbols. One of the reasons is that `get X` can be sequenced with an expression that makes use of the symbol Y .

6.3.2 Type inference

One of the main draws of the type system of ML_G is the fact that it admits a type inference algorithm, which is an extension of the algorithm given by Damas and

$$\begin{array}{l}
(\text{CONST}) \frac{\mathcal{T}_c \text{ is the base type of which } c \text{ is a member}}{\Sigma \mid \Delta \mid \Gamma \Vdash c : \mathcal{T}_c} \\
(\text{VAR}) \frac{}{\Sigma \mid \Delta \mid \Gamma \Vdash x : A} \text{ (if } (x : \sigma) \in \Gamma \text{ and } \sigma > A) \\
(\text{FN}) \frac{\Sigma \mid \Delta \mid x : A_1, \Gamma \Vdash M : A_2}{\Sigma \mid \Delta \mid \Gamma \Vdash \text{fn } x \Rightarrow M : A_1 \rightarrow A_2} \\
(\text{APP}) \frac{\Sigma \mid \Delta \mid \Gamma \Vdash M_1 : A_1 \rightarrow A_2 \quad \Sigma \mid \Delta \mid \Gamma \Vdash M_2 : A_1}{\Sigma \mid \Delta \mid \Gamma \Vdash M_1 M_2 : A_2} \\
(\text{LET}) \frac{\Sigma \mid \Delta \mid \Gamma \Vdash M_1 : A_1 \quad \Sigma \mid \Delta \mid x : (\forall_{\text{eff}} R)(\forall_{\text{type}} V) A_1, \Gamma \Vdash M_2 : A_2}{\Sigma \mid \Delta \mid \Gamma \Vdash \text{let } x = M_1 \text{ in } M_2 : A_2} \left(\begin{array}{l} x \notin \text{dom}(\Gamma) \\ V = \text{ftv}(A_1) \setminus \text{ftv}(\Gamma) \\ R = \text{fev}(A_1) \setminus \text{fev}(\Gamma) \end{array} \right) \\
(\text{BIND}) \frac{\Sigma \mid \Delta \mid \Gamma \Vdash M_1 : T^{F_1} A_1 \quad \Sigma \mid \Delta \mid x : (\forall_{\text{eff}} R)(\forall_{\text{type}} V) A_1, \Gamma \Vdash M_2 : T^{F_2} A_2}{\Sigma \mid \Delta \mid \Gamma \Vdash \text{bind } x \Leftarrow M_1 \text{ with } M_2 : T^{F_1 \odot F_2} A_2} \left(\begin{array}{l} x \notin \text{dom}(\Gamma) \\ V = \text{ftv}(A_1) \setminus \text{ftv}(\Gamma) \\ R = \text{fev}(A_1) \setminus \text{fev}(\Gamma) \end{array} \right) \\
(\text{RETURN}) \frac{\Sigma \mid \Delta \mid \Gamma \Vdash M : A}{\Sigma \mid \Delta \mid \Gamma \Vdash \text{return } M : T^1 A} \quad (\text{SUB}) \frac{\Sigma \mid \Delta \mid \Gamma \Vdash M : A_1 \quad A_1 <: A_2}{\Sigma \mid \Delta \mid \Gamma \Vdash M : A_2} \\
(\text{SYM-NEW}) \frac{X, \Sigma \mid \Delta \mid \text{prune}(\Gamma) \Vdash M : \tau}{\Sigma \mid \Delta \mid \Gamma \Vdash \text{new } X \text{ within } M : \tau} \\
(\text{SYM-FN}) \frac{\Sigma \mid \delta, \Delta \mid \Gamma \Vdash M : A}{\Sigma \mid \Delta \mid \Gamma \Vdash \text{sym } \delta \Rightarrow M : \Pi_{\text{sym}} \delta (A)} \\
(\text{SYM-APP}) \frac{\Sigma \mid \Delta \mid \Gamma \Vdash M : \Pi_{\text{sym}} \delta (A) \quad \Sigma \mid \mathcal{S} \mid \Delta \Vdash_{\text{sym}} \Psi}{\Sigma \mid \Delta \mid \Gamma \Vdash M \Psi : A\{\Psi/\delta\}}
\end{array}$$

Figure 6.6. Type system for ML_G, defined by $\Sigma \mid \Delta \mid \Gamma \Vdash M : A$ entailments.

Milner [DM82], whose variants are used in ML dialects and in Haskell. This means that type inference serves the purpose of program analysis as well. This goal (decidable type inference) informs some of the design decisions of ML_G.

We first present the algorithm that infers the types (and hence grades appearing within them) in ML_G, which turns out to be a simple extension of various well known unification algorithms. We then prove its correctness.

There is one important concept needed to define the algorithm: *type and grade substitutions*. Note that this generalises the notion of substitution used on type schemes.

Definition 6.3. A *type and grade substitution* θ is a total function from type variables α to ML_G types and grade variables r to grades such that $\theta(\alpha) = \alpha$ and $\theta(r) = r$ all but finitely often.

We adopt the notation θA to represent the application of θ to A , where all free occurrences of type and grade variables in A are replaced according to θ . Next, $\theta\sigma$ substitutes free occurrences of type variables and grade variables, renaming the bound variables if necessary. Furthermore, we write $\theta\Gamma$ to mean ‘ Γ , with θ applied to all types and grades that form the context’. We also write $\theta_2\theta_1$ to represent composition of θ_2 and θ_1 .

The *infer*($\cdot$) function is defined on $\Sigma \mid \Delta \mid \Gamma \Vdash M$ and returns either a pair (θ, A) or **FAIL**. The former result, when returned, should satisfy $\Sigma \mid \Delta \mid \theta\Gamma \Vdash M : A$, whereas returning **FAIL** means that the expression M is not typeable in context $\Sigma \mid \Delta \mid \Gamma$. In other words, *infer*($\cdot$) finds a solution to the typing problem for an expression M in a particular context. We make use of **fresh** (which creates a fresh name for a type or grade variable), **assertTau** (which asserts that a type is in τ , as defined in Section 6.3.1) and *unify* (which returns the most general unifier of two expressions). The result **FAIL** is produced whenever an assertion fails or the unification algorithm finds that no solution exists.

The type inference procedure *infer*($\cdot$) (see Fig. 6.7) works as expected: it returns a valid type scheme. Furthermore, it returns the most general solution to the typing problem – that is, the principal type scheme.

Theorem 6.2. The *infer*($\cdot$) function returns the *principal* type scheme for an expression in a given context, meaning:

1. for $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M) = (\theta, A)$, it is the case that $\Sigma \mid \Delta \mid \theta\Gamma \Vdash M : A$;
 2. the resulting type scheme $\sigma = (\forall_{\text{eff}} R)(\forall_{\text{type}} V)A$, where $R = \text{fev}(A) \setminus \text{fev}(\Gamma)$ and $V = \text{ftv}(A) \setminus \text{ftv}(\Gamma)$, is principal: for every type A' such that $\Delta \mid \Gamma \Vdash M : A'$, there exists a type A'' such that $\Sigma \mid \Delta \Vdash A'' <: A'$ and $\sigma > A''$.
-

Proof. See Appendix F.2. □

6.3.3 Relationship with System G

We did not define a semantics for ML_G because it is intended as a ‘surface language’ for System G: in other words, ML_G as a simplified version of System G with decidable type inference. This idea is expressed in the form of a *translation* of well-typed ML_G terms to well-typed System G terms.

We thus introduce the following translation function (defined on derivations of typing judgements) that maps ML_G typing judgements to corresponding System G judgements:

$$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M : A \rangle\rangle = M'$$

where the symbols $\Sigma, \Delta, \Gamma, M, A$ refer to ML_G concepts and the name M' refers to a System G term. The above application of $\langle\langle \Sigma, \Delta, \Gamma \Vdash M : A \rangle\rangle$ is only valid when its arguments satisfy $\Sigma \mid \Delta \mid \Gamma \Vdash M : A$ in ML_G and when A contains no type variables – it is fully monomorphised. In addition, we substitute all **let** $x = M$ **in** N with $N\{M/x\}$ so that the expression is fully monomorphised prior to translation. Type variables are omitted from our translation as System G has no polymorphic types.

(Here we use fresh sets (α -rename) for A and R at each use of x)	
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash x)$	$= \text{let } (\forall_{\text{eff}} R)(\forall_{\text{type}} V) A = \Gamma(x) \text{ in } (Id, A)$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash c)$	$= (Id, \mathcal{T}_c) \quad (\mathcal{T}_c \text{ is the type of } c)$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash \text{fn } x \Rightarrow M)$	$= \text{let } \alpha = \text{fresh in}$ $\text{let } (\theta, A) = infer(\Sigma \mid \Delta \mid x : \alpha, \Gamma \Vdash M) \text{ in}$ $(\theta, \theta(\alpha) \rightarrow A)$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash M_1 M_2)$	$= \text{let } \alpha = \text{fresh in}$ $\text{let } (\theta_1, A_1) = infer(\Sigma \mid \Delta \mid \Gamma \Vdash M_1) \text{ in}$ $\text{let } (\theta_2, A_2) = infer(\Sigma \mid \Delta \mid \theta_1 \Gamma \Vdash M_2) \text{ in}$ $\text{let } \theta_3 = unify(\theta_2 A_1, A_2 \rightarrow \alpha) \text{ in}$ $(\theta_3 \theta_2 \theta_1, \theta_3(\alpha))$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash \text{let } x = M_1 \text{ in } M_2)$	$= \text{let } (\theta_1, A_1) = infer(\Sigma \mid \Delta \mid \Gamma \Vdash M_1) \text{ in}$ $\text{let } R = fev(A_1) \setminus fev(\theta_1 \Gamma) \text{ in}$ $\text{let } V = ftv(A_1) \setminus ftv(\theta_1 \Gamma) \text{ in}$ $\text{let } \sigma = (\forall_{\text{eff}} R)(\forall_{\text{type}} V) A_1 \text{ in}$ $\text{let } (\theta_2, A_2) =$ $infer(\Sigma \mid \Delta \mid x : \sigma, \theta_1 \Gamma \Vdash M_2) \text{ in}$ $(\theta_2 \theta_1, A_2)$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash \text{new } X \text{ within } M)$	$= \text{let } (\theta, A) =$ $infer(X, \Sigma \mid \Delta \mid \text{prune}(\Gamma) \Vdash M) \text{ in}$ $\text{assertTau}(A);$ (θ, A)
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash \text{bind } x \Leftarrow M_1 \text{ with } M_2)$	$= \text{let } (\theta_1, A_1) = infer(\Sigma \mid \Delta \mid \Gamma \Vdash M_1) \text{ in}$ $\text{let } (\theta_2, A_2) =$ $infer(\Sigma \mid \Delta \mid x : A_1, \theta_1 \Gamma \Vdash M_2) \text{ in}$ $\text{let } \alpha = \text{fresh in let } \beta = \text{fresh in}$ $\text{let } r = \text{fresh in let } s = \text{fresh in}$ $\text{let } \theta_3 = unify(T^r \alpha, \theta_2 A_1) \text{ in}$ $\text{let } \theta_4 = unify(T^s \beta, \theta_3 A_2) \text{ in}$ $(\theta_4 \theta_3 \theta_2 \theta_1, T^{\theta_4 \theta_3(r) \odot \theta_4(s)}(\theta_4(\beta)))$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash \text{return } M)$	$= \text{let } (\theta, A) = infer(\Sigma \mid \Delta \mid \Gamma \Vdash M) \text{ in}$ $(\theta, T^1 A)$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash \text{sym } \delta \Rightarrow M)$	$= \text{let } (\theta, A) = infer(\Sigma \mid \delta, \Delta \mid \Gamma \Vdash M) \text{ in}$ $(\theta, \Pi_{\text{sym}} \delta(A))$
$infer(\Sigma \mid \Delta \mid \Gamma \Vdash M \Psi)$	$= \text{let } (\theta_1, A_1) = infer(\Sigma \mid \Delta \mid \Gamma \Vdash M) \text{ in}$ $\text{let } \alpha = \text{fresh in let } \delta = \text{fresh in}$ $\text{let } \theta_2 = unify(\Pi_{\text{sym}} \delta(\alpha), A_1) \text{ in}$ $(\theta_2 \theta_1, \theta_2(\alpha)\{\Psi/\delta\})$

Figure 6.7. The type inference algorithm for ML_G.

$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash c : A \rangle\rangle$	$= c$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash x : A \rangle\rangle$	$= x$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash \text{fn } x \Rightarrow M : A \rightarrow B \rangle\rangle$	$= \lambda x : \langle A \rangle_{[\Sigma]} (\langle\langle \Sigma \mid \Delta \mid x : A, \Gamma \Vdash M : B \rangle\rangle)$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M_1 M_2 : A \rangle\rangle$	$= \langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M_1 : B \rightarrow A \rangle\rangle$ $\quad \langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M_2 : B \rangle\rangle$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash \text{let } x = M_1 \text{ in } M_2, A \rangle\rangle$	$= \text{case omitted (see text)}$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash \text{new } X \text{ within } M : \tau \rangle\rangle$	$= \nu X (\langle\langle X, \Sigma \mid \Delta \mid \Gamma \Vdash M : A \rangle\rangle)$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash \text{sym } \delta \Rightarrow M, \Pi_{\text{sym}} \delta (A) \rangle\rangle$	$= \lambda_{\text{sym}} \delta : \Sigma (\langle\langle \Sigma \mid \delta, \Delta \mid \Gamma \Vdash M : A \rangle\rangle)$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M \Psi : A \rangle\rangle$	$= \langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M : \Pi_{\text{sym}} \delta (A) \rangle\rangle \Psi$ $\quad (\text{assuming } \Sigma \mid \Delta \mid \Gamma \Vdash M : \Pi_{\text{sym}} \delta (A))$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash \text{return } M : T^1 A \rangle\rangle$	$= \text{ret}_{T^\bullet \Sigma} \langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M : A \rangle\rangle$
$\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash \text{bind } x \Leftarrow M_1 \text{ with } M_2 : T^{F \odot G} B \rangle\rangle$	$= \langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M_1 : T^F A \rangle\rangle \gg_{=T^\bullet \Sigma} \lambda x : A (\langle\langle \Sigma \mid \Delta \mid \Gamma \Vdash M_2 : T^G B \rangle\rangle)$
$\langle \text{BaseTypes} \rangle_{[\Sigma]}$	$= \text{BaseTypes}$
$\langle A_1 \rightarrow A_2 \rangle_{[\Sigma]}$	$= \langle A_1 \rangle_{[\Sigma]} \rightarrow \langle A_2 \rangle_{[\Sigma]}$
$\langle T^F A \rangle_{[\Sigma]}$	$= T^\bullet \Sigma F \langle A \rangle_{[\Sigma]}$
$\langle \Pi_{\text{sym}} \delta (A) \rangle_{[\Sigma]}$	$= \Pi_{\text{sym}} \delta : \Sigma \langle A \rangle_{[\Sigma]}$

Figure 6.8. Translation of ML_G terms $\langle\langle \Sigma, \Delta, \Gamma \Vdash M : A \rangle\rangle$ and types $\langle A \rangle_{[\Sigma]}$ to System G. Implicit in the presentation of the first translation is that it is guided by the most recently applied rule in the typing derivation, and that the last rule applied is not (SUB). Subtyping is translated by performing a coercion that follows the derivation of $\Sigma \mid \Delta \Vdash A <: B$, inserting System G $\text{sub}_{T^\bullet \Sigma} F G M$ terms when needed.

The full definition of $\langle\langle \Sigma, \Delta, \Gamma \Vdash M : A \rangle\rangle$ is stated in Figure 6.8. The main interesting point is adding explicit symbol sets to the context Δ and explicitly stating the symbol set when graded monadic operations are used. To avoid type polymorphism, local definitions via `let  $x = M$  in  $N$` are translated by translating the equivalent term $N\{M/x\}$, with the intention that polymorphism is avoided by using the appropriate monomorphisation of M in all instances in which it is used. Furthermore, we use a $\langle \cdot \rangle_{[\Sigma]}$ function that maps ML_G types into System G types. This translation is necessary to translate symbol abstractions from ML_G to System G, where the symbol set has to be specified – and we make the choice that the set over which symbol variables range is just Σ .

This translation maps well-typed ML_G terms to well-typed System G terms, as stated by the following theorem.

Theorem 6.3. For all $\Sigma, \Delta, \Gamma, M, A$ satisfying $\Sigma \mid \Delta \mid \Gamma \Vdash M : A$ in ML_G , with A containing no type variables, the judgement $\Sigma \mid \bullet \mid \Delta' \mid \bullet \mid \Gamma' \vdash_{\text{val}} M' : A'$ holds in System G, where: $\langle\langle \Sigma, \Delta, \Gamma \Vdash M : A \rangle\rangle = M'$, $\langle A \rangle_{[\Sigma]} = A'$ and where Δ' is just Δ in which all symbol variables δ are replaced with $\delta : \Sigma$, and Γ' is just the context Γ in which all typing pairs $x : B$ are replaced by $x : \langle B \rangle_{[\Sigma]}$.

Technically, the definition of Δ' may increase the set of symbols which δ can range over, but the translation into System G remains valid.

This means that the relationship between ML_G and System G is similar to that between Standard ML (without recursion) and System F: in a sense, we can see the latter as a kernel language on whose semantics the former relies on.

6.4 Summary

In this chapter we looked at the concept of symbols and how they are used for programming and representing program analyses. We considered two languages, ML_G and System G, which approach the question of lexically scoped symbols (and scoped graded monadic types) in two different ways: ML_G has decidable type inference, but does not permit two graded monadic types to coexist in the same scope; System G has no type inference algorithm and instead takes the System F approach of explicitly stating all types, but it allows greater flexibility regarding the interaction of multiple graded monadic types in the same scope.

Designing a language with support for symbols is promising and is an important component of making programming with graded monads practical for common analyses.

Combining analyses

So far in this thesis the focus has been on programs in which graded monadic types $M^F A$ correspond to only one kind of effect and only one analysis. The running example of the graded `Maybe` monad has a three-element grading algebra representing success and failure of a computation, and we considered various grading algebras that can be used to reason about stateful computations. In imperative programming languages we encounter programs with multiple kinds of side-effects: they may act on mutable store and also raise and handle exceptions. Similarly, the programmer may be interested in performing multiple analyses at the same time. The main goal of this chapter is to explore how this can be achieved and modelled.

In a pure functional programming language, where effects are modelled using monads, the idiomatic way of combining effects is by using monad transformers. A monad transformer t is just a constructor that maps a monad m representing some kind of computation to another monad $t\ m$ representing a “more sophisticated” computation. The `MaybeT` monad transformer in Haskell, when applied to some monad m , creates a monadic type that retains all the functionality m provides, but also allows the computation to fail. For example, `MaybeT (State s) a` represents a stateful computation (with an underlying store of type s) which, if it terminates, reduces to a value of type a or fails. An important component of every monad transformer is a lifting operation `lift :: m a -> t m a` that takes a value of the “inner” monadic type $m\ a$ and represents it as a computation in the transformed monad $t\ m\ a$.

Monad transformers are a valuable tool for pure functional programmers. However, there is a surprisingly small body of work about the underlying theory of monad transformers, with the most significant contribution coming from Jaskelioff in his PhD thesis and his work with Moggi. [Jas09a, JM10b] I will give a brief overview of his work later in this chapter.

In the graded world, even adequately posing the the question of what it means to combine two graded monads is more difficult. This thesis explicitly considers analysis-graded monads, meaning that a graded monadic type represents a computation along with pertinent information that represents the outcome of some program analysis. Therefore, to combine two graded monads, we need to answer the question of what it means to combine two analyses. In the graded setting, we go beyond the existing approach adopted in Haskell and discussion on combining effects. We distinguish between two ways of combining analyses: combining two analyses that refer to different notions of computation and combining analyses which relate to the same side-effect. I address the former case using *graded monad transformers*, which adapt the definition of classical non-graded monad transformers by adding both a new side-effect and a new accompanying analysis. The latter case is answered using the notion of *grade families*, which are a way of defining a set of

grading algebras and graded monads that address the same effect and allowing the programmer to combine them at will.

Chapter structure In Section 7.1 I define graded monad transformers – first from a programming point of view, and then formally in categorical terms. It turns out that graded monad transformers can be implemented in Haskell (encoding grades as type-level values, as before), and the core text of this chapter sketches how this is done (with details given in the Appendix). The category-theoretic view mostly just provides definitions and formal terminology, as well as addresses some fundamental questions about the nature of graded monad transformers. In Section 7.2 I discuss grade families – which are much harder to use in a practical programming language. Next, in Section 7.3 I discuss the limitations of these two approaches and how these two ways of combining analyses interact with each other. The chapter ends with an overview of related work and a summary.

7.1 Graded monad transformers

We first motivate what we expect of graded monad transformers before exploring them in the context of programming, formalising them and proving some useful results related to them. The initial treatment of graded monad transformers is informal.

Recall that a monad transformer maps a monad to some other monad, and in principle the definition could stop there. However, we typically look at monad transformers that add a new side effect while maintaining the possibility of performing the one for the ‘inner’ monad. A monad transformer R does not merely create a new underlying representation and monad $R[T]$, but it also comes with a lift operation that maps an expression of type TA to one of type $(R[T])A$.

Therefore, a graded monad transformer should act on a graded monad and produce a new graded monad, and it should come with a graded lifting operation. While in the most general case we can stop with the definition there, we really focus on those graded monad transformers that are somehow *useful*. The grading algebra to be associated with the resulting graded monad is not immediately obvious. Intuitively, the a graded monad transformer should add a new side effect and a new analysis. Since analyses are represented using grading algebras, our first attempt is to merely take the product of the algebra associated with the transformer and the algebra associated with the input graded monad. Taking a cartesian product algebra means that the graded bind operation treats the two side effects and analyses independently.

Unfortunately, this approach is not satisfactory. Consider again the **GMaybe** graded monad whose grading algebra represents the analysis of whether a computation provably fails or succeeds, initially introduced in Example 2.2. Recall that its grading algebra is $(\{\mathbf{T}, \mathbf{F}, \mathbf{U}\}, \sqsubseteq, \odot, \mathbf{T}, \{\mathbf{T}, \mathbf{F}\})$ and that $\mathbf{F} \odot r = \mathbf{F}$ for all $r \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}$. A graded monad transformer **GMaybeT** corresponding to **GMaybe** would then map a graded monad with a grading algebra $\mathbb{E} = (E, \leq, *, \mathbf{1})$ to a graded monad whose grading algebra is:

$$(\{\mathbf{T}, \mathbf{F}, \mathbf{U}\}, \sqsubseteq, \odot, \mathbf{T}, \{\mathbf{T}, \mathbf{F}\}) \times (E, \leq, *, \mathbf{1}).$$

If the new operator is $\odot = \odot \times *$, then for any $x, y \in E$ and $r \in \{\mathbf{T}, \mathbf{F}, \mathbf{U}\}$ we have that $\langle \mathbf{F}, x \rangle \odot \langle r, y \rangle = \langle \mathbf{F}, x * y \rangle$. However, this does not match the desired outcome:

if a computation c_1 provably fails, then sequencing it with any other computation c_2 should ignore the outcomes of all of its analyses, as c_2 is never actually executed. We want the outcome of multiplying $\langle F, x \rangle$ with anything else to be $\langle F, x \rangle$.

The product algebra is a bad choice because of the way the two analyses interact and the sorts of side effects they represent. There is nothing problematic about making the carrier set of the resulting grading algebra the cartesian product of the two constituent algebras, as well as using componentwise subeffecting. The issue is solely with the new monoid multiplication operator. By imposing the condition that the grading algebra has a carrier set that is a cartesian product, but has no such requirements on the multiplication operator, we see what a ‘useful’ graded monad transformer is. From now on we call them *augmenting graded monad transformers*.

It remains to determine what the lifting operation for an augmenting graded monad transformer should be. Suppose the grading algebra associated with a transformer R is $\mathcal{E}_1 = (E_1, \leq_1, \odot_1, \mathbf{1}_1, \mathbf{O}_1)$, while the grading algebra associated with a graded monad T is $\mathcal{E}_2 = (E_2, \leq_2, \odot_2, \mathbf{1}_2, \mathbf{O}_2)$. Lifting should map a value of type $T^e A$ to one of type $(R[T])^{e'} A$, where $e \in E_2$ and $\langle f, e' \rangle \in E_1 \times E_2$. As is the case in the non-graded setting, applying the lifting operation should not perform any non-trivial computation associated with R (‘outer level’), so the $f = \mathbf{1}_1$. If e' is always e , then the transformer preserves the outcome of the original analysis – we then refer to it as *analysis-preserving*. There are useful graded monad transformers that are not analysis-preserving. However, it must be the case that $e \leq_2 e'$ because the resulting grade must be a conservative overapproximation of the property being analysed.

7.1.1 Haskell view

As is the case with programming with graded monads in general, a language with support for graded monad transformers needs to have a flexible kind system so that the carrier set of every grading algebra has a corresponding kind. In previous chapters we have seen that Haskell is such a language. In this section we demonstrate how graded monad transformers can be represented in Haskell using existing language features.

We only consider analysis-preserving graded monad transformers. Thus, every graded monad transformer must be an instance of the following simple type class that implements a graded lifting operation:

```
class GMonadTrans (t :: (k -> * -> *) -> (k', k) -> * -> *)
                                where
    type IdG t :: k'
    liftG      :: GMonad m => m s a -> t m '(IdG t, s) a
```

where the kinds k and k' are polymorphic in the type class signature, and $*$ is the kind of types. The kind of a graded monad transformer is

```
(k -> * -> *) -> (k', k) -> * -> *
```

which means that it maps a graded monad whose grading algebra is represented as kind k to a graded monad whose underlying grading algebra is represented as the pair kind (k', k) . The kind k' is associated with the grading algebra that represents the analysis added by the transformer. The `IdG t` term of kind k' is, in all scenarios, the identity element of the analysis being added by the transformer t . The type signature of the lifting operation `liftG` exactly matches what we expect from analysis-preserving graded monad transformers.

Every graded monad transformer implemented in Haskell should, when applied to a graded monad, result in a graded monad. Therefore, we need to write a following instance for each graded monad transformer:

```
instance GMonad m => GMonad (MyFancyGMonadTrans m) where
  ...
```

for every graded monad transformer `MyFancyGMonadTrans`.

Implementing graded monad transformers sometimes requires writing a lot of code. Therefore this section does not contain any lengthy listings with graded monad transformer implementations. However, the following example demonstrates both how graded monad transformers are used in practice and how they are implemented.

Example 7.1. Consider the graded monad transformer `GMaybeT` that adds failure as a possible effect and whose grade represents provable success or provable failure. As seen above, given a graded monad with underlying algebra $(E, \leq, *, 1)$, the algebra of pairs with operator $\odot$ should be defined as follows:

$$\begin{aligned} \langle T, x \rangle \odot \langle r, y \rangle &= \langle r, x * y \rangle \\ \langle F, x \rangle \odot \langle r, y \rangle &= \langle F, x \rangle \\ \langle U, x \rangle \odot \langle F, y \rangle &= \langle F, x * (1 \vee y) \rangle \\ \langle U, x \rangle \odot \langle r, y \rangle &= \langle U, x * (1 \vee y) \rangle \quad (r \neq F) \end{aligned}$$

The underlying representation of a graded monad `GMaybeT m` follows that of `MaybeT`:

```
data GMaybeT m r a where
  GMaybeT :: {runGMaybeT :: m f (GMaybe e a)}
             -> GMaybeT m '(e, f) a
```

The multiplication operator, following the definition above, is implemented as the following type family:

```
type family MaybeTMult (m :: k -> * -> *)
                      (r :: (MaybeGr, k))
                      (s :: (MaybeGr, k)) where
  MaybeTMult m '(F, f) _ = '(F, f)
  MaybeTMult m '(T, f) '(s, f') = '(s, Seq m f f')
  MaybeTMult m '(U, f) '(F, f') =
    '(F, Seq m f (Join m f' (Unit m)))
  MaybeTMult m '(U, f) '(_, f') =
    '(U, Seq m f (Join m f' (Unit m)))
```

Showing that this is really a graded monad transformer is not difficult, as the instances are straightforward:

```
instance GMonad m =>
  GMonad (GMaybeT m :: (MaybeGr, k) -> * -> *) where
  type Unit (GMaybeT m) = '(T, Unit m)
  type Seq (GMaybeT m) r s = MaybeTMult m r s
  type Join (GMaybeT m) '(e, f) '(e', f') =
    '(MaybeJoin e e', Join m f f')
  return x = GMaybeT (return (GJust x))
instance GMonadTrans GMaybeT where
```

```

type IdG GMaybeT = 'T
liftG = GMaybeT . (fmapG GJust)

```

The real challenge is implementing the bind operation for this graded monad, and details of how this is done are in the Appendix.

With this implementation, the programmer can write code utilising the `GMaybeT` graded monad transformer. For example, the following function represents a computation that is both stateful and potentially failing:

```

foo = do x <- liftG get
      alt (x == 0) (GMaybeT (return GNothing))
          (liftG (put 2))

```

and the Haskell type system is capable of inferring the type of this computation, which includes the grade `U`. On the other hand, it is also capable of inferring that the following computation also fails, which is seen in the presence of grade `e`:

```

bar x = do alt (x < 0) (GMaybeT (return GNothing))
          (liftG (put (x + 1)))
        GMaybeT (return GNothing)

```

The Appendix contains code listings giving a full definition of several useful graded monad transformers. In addition to `GMaybeT`, these include:

- `GrIdentityT`, which is the trivial graded monad transformer that adds no new effect and no analysis.
- `PermStateT s`, which is the graded monad transformer corresponding to the `PermState` graded monad – given a graded monadic type representing a computation and an analysis, it adds a state to the computation and permission information to the algebra.
- `LogStateT s`, which is the graded monad transformer corresponding to the `LogState s` graded monad that keeps track of all intermediate values of the underlying state.

7.1.2 Categorical definition

The Haskell implementation and the use cases demonstrate the value of graded monad transformers, but answering any theoretical questions about the scope and power of graded monad transformers requires a mathematical view. We now turn to a more formal, categorical treatment of graded monad transformers.

We start with a set of definitions that introduces the hierarchy of graded monad transformers suggested in previous sections.

Definition 7.1. A *graded monad transformer* on a category $\mathbb{C}$ is given by:

- an object mapping $R : |\mathbf{GMnd}(\mathbb{C})| \rightarrow \mathbf{GMnd}(\mathbb{C})^a$
- for each $\mathbf{T} : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$ there is a lax monoidal functor $H_{\mathbf{T}} : E \rightarrow R[E]$ (seen as monoidal categories), representing a monotonic map on grading algebras;

- for all graded monads $\mathbf{T} = (T, \mu, \eta)$, a monoidal natural transformation $\text{liftG}^{\mathbf{T}}$ with $\text{liftG}_e^{\mathbf{T}} : Te \rightarrow R[T](H_{\mathbf{T}}e)$.

^aThe notation $|\mathbb{C}|$ for a category $\mathbb{C}$ refers to a category with all the objects of $\mathbb{C}$, but the only morphisms are identities. Functors of the form $|\mathbb{C}| \rightarrow \mathbb{D}$ can be thought of as mappings of objects from $\mathbb{C}$ to $\mathbb{D}$, since functors preserve identities.

The first part of the definition means that R is just an object mapping. The functors $H_{\mathbf{T}}$ act as translations from the grading algebra associated with $\mathbf{T}$ to the one associated with $R[\mathbf{T}]$. As $f : e_1 \rightarrow e_2$ can really be read as $e_1 \leq e_2$, functoriality of $H_{\mathbf{T}}$ implies that it is a monotonic mapping. This last part of the definition is equivalent to saying that the following diagram commutes for all $f : e_1 \rightarrow e_2$:

$$\begin{array}{ccc} Te_1 & \xrightarrow{\text{liftG}_{e_1}^{\mathbf{T}}} & R[T](H_{\mathbf{T}}e_1) \\ Tf \downarrow & & \downarrow R[T](H_{\mathbf{T}}f) \\ Te_2 & \xrightarrow{\text{liftG}_{e_2}^{\mathbf{T}}} & R[T](H_{\mathbf{T}}e_2) \end{array}$$

which implies that lifting preserves subeffecting.

We denote graded monad transformers as tuples (R, H, liftG) . Furthermore, the application of this graded monad transformer to a graded monad $\mathbf{T} = (T, \mu, \eta) : \mathbb{E} \rightarrow \mathbf{Endo}(\mathbb{C})$, with $\mathbb{E} = (E, \leq, \odot, 1)$, results in the graded monad $R\mathbf{T} = (R[T], R[\mu], R[\eta]) : R[\mathbb{E}] \rightarrow \mathbf{Endo}(\mathbb{C})$, where the algebra is mapped as $R(\mathbb{E}) = (R[E], R[\leq], R[\odot], R[\eta])$.

In practice we often require or care about graded monadic strength. We thus formalise the idea of *strong graded monad transformers*, which are applied to and result in strong graded monads. Since the definition of a strong graded monad transformer might rely on strength of the graded monad it is applied to, it is not sufficient to say that a strong graded monad transformer is a graded monad transformer such that the resulting graded monad is strong. Instead, we have to modify the previous definition slightly.

Definition 7.2. A *strong graded monad transformer* on a category $\mathbb{C}$ is given by:

- an object mapping $R : |\mathbf{SGMnd}(\mathbb{C})| \rightarrow \mathbf{SGMnd}(\mathbb{C})$;
- for each $\mathbf{T} : (E, \leq, \odot, 1) \rightarrow \mathbf{SEndo}(\mathbb{C})$ there is a lax monoidal functor $H_{\mathbf{T}} : E \rightarrow R[E]$;
- for all strong graded monads $\mathbf{T} = (T, \mu, \eta, t)$, a monoidal natural transformation $\text{liftG}^{\mathbf{T}}$ with $\text{liftG}_e^{\mathbf{T}} : Te \rightarrow R[T](H_{\mathbf{T}}e)$.

Note that strong graded monad transformers are not a subset of graded monad transformers, as graded monad transformers R are not allowed to rely on the strength of $\mathbf{T}$ when defining multiplication and unit for $R\mathbf{T}$, as $\mathbf{T}$ might not be strong. The lack of this relation is not practically significant, as all analysis-graded monads used in a programming context are strong.

These definitions are very general and permit *forgetful* (strong) graded monad transformers that disregard the underlying analysis. A more useful notion is that of a graded monad transformer that constructs a new grading algebra of pairs.

Definition 7.3. A (strong) graded monad transformer $(R, H, \text{lift}G)$ on $\mathbb{C}$ is *augmenting* if there exists a poset $(S, \sqsubseteq)$ with a top element and $i \in S$ (which we call the *augmenting poset* and *augmenting unit*) such that for every (strong) graded monad $\mathbf{T} = (T, \mu, \eta) : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$,^a $R\mathbf{T}$ is a (strong) graded monad whose grading algebra has carrier set $S \times E$ with unit element is $\langle i, \mathbf{1} \rangle$, and for every $e \in E$ there exists some e' such that $e \leq e'$ and $H_{\mathbf{T}}e = \langle i, e' \rangle$.

^areplace $\mathbf{Endo}(\mathbb{C})$ by $\mathbf{SEndo}(\mathbb{C})$ in the strong case

The last condition ensures that lifting does not produce any side-effect for the added analysis by using its augmenting unit, while never underapproximating the result of the property associated with $\mathbf{T}$. In most cases we are interested in augmenting (strong) graded monad transformers which lifting keeps the information about the inner analysis intact.

Definition 7.4. An augmenting (strong) graded monad transformer $(R, H, \text{lift}G)$ with augmenting unit i is *analysis-preserving* such that, given any (strong) graded monad $\mathbf{T} : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$,^a and $e \in E$, $H_{\mathbf{T}}e = \langle i, e \rangle$.

^areplace $\mathbf{Endo}(\mathbb{C})$ by $\mathbf{SEndo}(\mathbb{C})$ in the strong case

Example 7.2. The identity graded monad transformer $\mathbf{GIdentityT}$ maps any graded monad $\mathbf{T} = (T, \mu, \eta) : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$ to a graded monad $\widehat{\mathbf{T}}$ whose grading algebra is the Cartesian product of the trivial one-element algebra $\mathbb{1}$ and $(E, \leq, \odot, \mathbf{1})$ (meaning it is isomorphic to $(E, \leq, \odot, \mathbf{1})$), with new componentwise partial order called $\leq$, and with a trivially lifted functor $\widehat{T}$, multiplication $\widehat{\mu}$ and unit $\widehat{\eta}$:

$$\begin{aligned} \widehat{T}\langle *, e \rangle &= Te \\ \widehat{T}(\langle *, e \rangle \leq \langle *, e' \rangle) &= T(e \leq e') \\ \widehat{\mu}_{\langle *, e \rangle, \langle *, e' \rangle} &= \mu_{e, e'} \\ \widehat{\eta} &= \eta \end{aligned}$$

The grade mapping is trivial: $H_{\mathbf{T}}e = \langle *, e \rangle$. Finally, the lifting natural transformation is given by $\text{lift}G_e^{\mathbf{T}} = \text{id}_{T_e}$ for all $e \in E$. It is easily shown that $\widehat{\mathbf{T}}$ is a graded monad. The $\mathbf{GIdentityT}$ graded monad transformer is analysis-preserving.

Example 7.3. The augmenting strong graded monad transformer $\mathbf{GMaybeT}$ in a cocartesian category $\mathbb{C}$ with terminal object $\mathbf{1}$ has as its augmenting poset $(\{\mathbf{T}, \mathbf{F}, \mathbf{U}\}, \sqsubseteq)$, with $\mathbf{T} \sqsubseteq \mathbf{U} \sqsupseteq \mathbf{F}$ (see Figure 2.6) and as its augmenting unit $\mathbf{T}$. It maps any graded monad $\mathbf{T} = (T, \mu, \eta) : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$ to a graded monad $\widehat{\mathbf{T}}$ whose grading algebra has as multiplication operator $\odot$, defined as follows:

$$\begin{aligned} \langle \mathbf{T}, e_1 \rangle \odot \langle r, e_2 \rangle &= \langle r, e_1 \odot e_2 \rangle \\ \langle \mathbf{F}, e_1 \rangle \odot \langle r, e_2 \rangle &= \langle \mathbf{F}, e_1 \rangle \\ \langle \mathbf{U}, e_1 \rangle \odot \langle \mathbf{F}, e_2 \rangle &= \langle \mathbf{F}, e_1 \odot (\mathbf{1} \vee e_2) \rangle \\ \langle \mathbf{U}, e_1 \rangle \odot \langle r, e_2 \rangle &= \langle \mathbf{U}, e_1 \odot (\mathbf{1} \vee e_2) \rangle \quad (r \neq \mathbf{F}) \end{aligned}$$

Denote the new componentwise partial order with $\leqslant$. The functor $\widehat{T}$ and natural transformations $\widehat{\mu}$ and $\widehat{\eta}$ are defined as follows:

$$\begin{aligned}\widehat{T}\langle \mathbf{T}, e \rangle &= Te \\ \widehat{T}\langle \mathbf{F}, e \rangle &= (Te)[!]\ \\ \widehat{T}\langle \mathbf{U}, e \rangle &= (Te)(\text{Id}_C + [!])\end{aligned}$$

$$\begin{aligned}\widehat{T}(\langle \mathbf{T}, e_1 \rangle \leqslant \langle \mathbf{T}, e_2 \rangle) &= T(e_1 \leqslant e_2) & \widehat{T}(\langle \mathbf{T}, e \rangle \leqslant \langle \mathbf{U}, e \rangle) &= \iota_1^{Te, (Te)[!]} \\ \widehat{T}(\langle \mathbf{F}, e_1 \rangle \leqslant \langle \mathbf{F}, e_2 \rangle) &= (T(e_1 \leqslant e_2))[!] & \widehat{T}(\langle \mathbf{F}, e \rangle \leqslant \langle \mathbf{U}, e \rangle) &= \iota_2^{Te, (Te)[!]} \\ \widehat{T}(\langle \mathbf{U}, e_1 \rangle \leqslant \langle \mathbf{U}, e_2 \rangle) &= T(e_1 \leqslant e_2)(\text{Id}_C + [!])\end{aligned}$$

(other inequalities follow by composition)

$$\begin{aligned}(\widehat{\mu}_{\langle \mathbf{T}, e_1 \rangle, \langle \mathbf{T}, e_2 \rangle})_A &= (\mu_{e_1, e_2})_A \\ (\widehat{\mu}_{\langle \mathbf{T}, e_1 \rangle, \langle \mathbf{F}, e_2 \rangle})_A &= (\mu_{e_1, e_2})_1 \\ (\widehat{\mu}_{\langle \mathbf{T}, e_1 \rangle, \langle \mathbf{U}, e_2 \rangle})_A &= (\mu_{e_1, e_2})_{A+1} \\ (\widehat{\mu}_{\langle \mathbf{F}, e_1 \rangle, \langle \mathbf{F}, e_2 \rangle})_A &= \text{id}_{Te_1 1} \\ (\widehat{\mu}_{\langle \mathbf{U}, e_1 \rangle, \langle \mathbf{F}, e_2 \rangle})_A &= (\mu_{e_1, 1 \vee e_2})_1 \circ Te_1[\text{id}_{T(1 \vee e_2) 1}, \text{id}_{T(1 \vee e_2) 1}] \\ &\quad \circ Te_1((T(e_2 \leqslant 1 \vee e_2))_A + ((T(1 \leqslant 1 \vee e_2))_1) \circ \eta_1)) \\ (\widehat{\mu}_{\langle \mathbf{U}, e_1 \rangle, \langle \mathbf{T}, e_2 \rangle})_A &= (\mu_{e_1, 1 \vee e_2})_{A+1} \circ Te_1[\text{id}_{T(1 \vee e_2)_{A+1}}, \text{id}_{T(1 \vee e_2)_{A+1}}] \\ &\quad \circ Te_1(((T(e_2 \leqslant 1 \vee e_2))_{A+1} \circ Te_2 \iota_1^{A, 1}) \\ &\quad + ((T(1 \leqslant 1 \vee e_2))_{A+1} \circ T1 \iota_2^{A, 1} \circ \eta_1)) \\ (\widehat{\mu}_{\langle \mathbf{U}, e_1 \rangle, \langle \mathbf{U}, e_2 \rangle})_A &= (\mu_{e_1, 1 \vee e_2})_{A+1} \circ Te_1[\text{id}_{T(1 \vee e_2)(A+1)}, \text{id}_{T(1 \vee e_2)(A+1)}] \\ &\quad \circ Te_1((T(e_2 \leqslant 1 \vee e_2))_{A+1} \\ &\quad + ((T(1 \leqslant 1 \vee e_2))_{A+1} \circ T1 \iota_2^{A, 1} \circ \eta_1))\end{aligned}$$

$$\widehat{\eta} = \eta$$

It can be shown that $\widehat{T}$ is a graded monad.

The grade mapping is trivial: $H_{Te} = \langle \mathbf{T}, e \rangle$. This means that the GMaybeT graded monad transformer is analysis-preserving. Finally, the lifting operation is $\text{lift}_e^T = \text{id}_{Te}$ for all $e \in E$.

The definition of multiplication $\widehat{\mu}$ is somewhat difficult to read and is easier explained with a diagram. We make use of the unit η and coproduct injections to coerce 1 into $T1(A+1)$, and then use subeffecting to coerce Te_2 and $T1$ into $T(1 \vee e_2)$. For example, $\widehat{\mu}_{\langle \mathbf{U}, e_1 \rangle, \langle \mathbf{U}, e_2 \rangle}$ is the unique natural transformation for which the following diagram commutes for all $A \in \mathbb{C}$:

$$\begin{array}{ccc} Te_1(Te_2(A+1) + 1) & \xrightarrow{Te_1(\text{id}_{Te_2(A+1)} + (T1 \iota_2^{A, 1} \circ \eta_1))} & Te_1(Te_2(A+1) + T1(A+1)) \\ \downarrow (\widehat{\mu}_{\langle \mathbf{U}, e_1 \rangle, \langle \mathbf{U}, e_2 \rangle})_A & & \downarrow Te_1(T(e_2 \leqslant 1 \vee e_2)(A+1) + (T(1 \leqslant 1 \vee e_2))_{A+1}) \\ & & Te_1(T(1 \vee e_2)(A+1) + T(1 \vee e_2)(A+1)) \\ & & \downarrow Te_1[\text{id}_{T(1 \vee e_2)(A+1)}, \text{id}_{T(1 \vee e_2)(A+1)}] \\ T(e_1 \odot (1 \vee e_2))(A+1) & \xleftarrow{\mu_{e_1, 1 \vee e_2}} & Te_1(T(1 \vee e_2)(A+1)) \end{array}$$

We omit the proofs that the result RT in each case is an analysis-graded monad – this requires showing that the new multiplication and unit natural transformations satisfy the associativity and unit laws, that the strength properties for the new graded monad hold, that H_T is a lax monoidal functor, as well as that all the

naturality diagrams commute. The proofs of these properties are outlined in the Appendix.

7.1.3 Properties

Graded monad transformers are introduced with the same goal as monad transformers in the non-graded setting: introducing a new effect to a computation. Naturally, this gives rise to questions about the the exact power and scope of graded monad transformers.

- Do graded monad transformers generalise monad transformers?
- Is there a monad transformer corresponding to every graded monad transformer?
- Are there other notions of combining graded monads and how do they relate to graded monad transformers?

We address these questions in turn and draw some further conclusions.

Relating monad transformers and graded monad transformers

The fact that monads are graded monads with the trivial grading algebra ($\mathbb{1}$) does not immediately show that every monad transformer is a graded monad transformer: monad transformers expects any monad as an argument, whereas a graded monad transformer should expect any graded monad – a stronger requirement. It turns out that this is not a problem – but only because of our hierarchy of graded monad transformers:

$$\text{transformer} \supseteq \text{augmenting} \supseteq \text{analysis-preserving}$$

In fact, each graded monad transformer (R, lift) has a corresponding augmenting graded monad transformer (R_g, H, liftG) – but it is *not* analysis-preserving.

Theorem 7.1. For every monad transformer (R, lift) in a category $\mathbb{C}$, there is a graded monad transformer (R_g, H, liftG) that maps any analysis-graded monad $\mathbf{T} = (T, \mu, \eta) : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$ whose algebra has a top element $\top$ to an analysis-graded monad $R_g \mathbf{T}$ as follows:

- the effect algebra is trivial, $\mathbb{1}$ (with only $*$ $\in \mathbb{1}$);
- $H_{\top}e = *$ for any $e \in E$;
- assuming $\mu^\top$ and $\eta^\top$ are the multiplication and unit natural transformations for the monad $T\top$, the components of the graded monad $R_g \mathbf{T}$ definition are as follows:

$$\begin{aligned} (R_g[T])^* &= R[T\top] \\ (R_g[\mu])_{*,*} &= R[\mu^\top] \\ (R_g[\eta]) &= R[\eta^\top] \end{aligned}$$

- the graded lifting operation erases information about the underlying analysis:

$$\text{lift}G_e^T = \text{lift}_{T\top} \circ T(e \leq \top)$$

Proof. The resulting $R_g T$ is a graded monad representing a monad. This follows from the assumption that R is a monad transformer and the fact that $(T\top, \mu^\top, \eta^\top)$ is a monad (Theorem 5.1). Furthermore, H_T is trivially a lax monoidal functor (as is any monoidal functor to $\mathbb{1}$). Finally, $\text{lift}G^T$ is a monoidal natural transformation. Firstly, for every $e \in E$, $\text{lift}G_e^T$ is a natural transformation – for all $f : A \rightarrow B$, the following diagram commutes (by naturality of $T(e \leq \top)$ and $\text{lift}_{T\top}$):

$$\begin{array}{ccccccc} TeA & \xrightarrow{(T(e \leq \top))_A} & T\top A & \xrightarrow{\text{lift}_{T\top A}} & (R[T\top])A & \xlongequal{\quad} & (R_g[T])(H_T e)A \\ \text{\scriptsize } Tef \downarrow & & \downarrow T\top f & & \downarrow R[T\top]f & & \downarrow (R_g[T])(H_T e)f \\ TeB & \xrightarrow{(T(e \leq \top))_B} & T\top B & \xrightarrow{\text{lift}_{T\top B}} & (R[T\top])B & \xlongequal{\quad} & (R_g[T])(H_T e)B \end{array}$$

Lifting respects the multiplication and unit laws, as the following diagrams commute by naturality and properties of monad transformer (R, lift) :

$$\begin{array}{ccccc} (Te_1) \bullet (Te_2) & \xrightarrow{(T(e_1 \leq \top)) \bullet (T(e_2 \leq \top))} & (T\top) \bullet (T\top) & \xrightarrow{\text{lift}_{T\top} \bullet \text{lift}_{T\top}} & (R[T\top]) \bullet (R[T\top]) \\ \mu_{e_1, e_2} \downarrow & & \downarrow \mu_{\top, \top} = \mu^\top & & \downarrow R[\mu^\top] \\ T(e_1 \odot e_2) & \xrightarrow{T(e_1 \odot e_2 \leq \top)} & T\top & \xrightarrow{\text{lift}_{T\top}} & R[T\top] \end{array}$$

$$\begin{array}{ccccc} & & \text{\textcolor{red}{Id}}_{\mathbb{C}} & & \\ & \eta \swarrow & \downarrow \eta^\top & \searrow R[\eta] & \\ T\mathbf{1} & \xrightarrow{T(1 \leq \top)} & T\top & \xrightarrow{\text{lift}_{T\top}} & R[T\top] \end{array}$$

□

This theorem states that every monad transformer can be seen as a graded monad transformer, but not a particularly useful one – it entirely removes the underlying grading algebra of the graded monad applied to the graded monad transformer.

Conversely, for every graded monad transformer there is a corresponding monad transformer one gets by ‘forgetting’ the underlying analysis.

Theorem 7.2. For every graded monad transformer $(R, H, \text{lift}G)$ in a category $\mathbb{C}$, there is monad transformer (R_u, lift) such that, given any monad $T = (T, \mu, \eta)$, if the trivially-graded monad equivalent to T is $\widehat{T}$, and the top element of the grading algebra corresponding to $R\widehat{T}$ is $\top$, gives the following monad $R_u T$:

$$\begin{aligned} R_u[T] &= (R[\widehat{T}])\top \\ R_u[\mu] &= (R[\widehat{\mu}])_{\top, \top} \\ R_u[\eta] &= R[\widehat{T}](1 \leq \top) \circ R[\widehat{\eta}] \end{aligned}$$

with the lifting operation given by:

$$\text{lift}^T = R[\widehat{T}](H * \leq \top) \circ \text{lift} G^{\widehat{T}}$$

Proof. Since $\mathbf{T}$ is a monad, and thus $\widehat{\mathbf{T}}$ is a graded monad, the result of applying R to it is $R\widehat{\mathbf{T}}$, itself a graded monad. Taking its top element, as stated in Theorem 5.1, gives a monad. All of the monad laws for the result $R_u\mathbf{T}$ hold.

The new lifting operations is a natural transformation, as the following diagram commutes:

$$\begin{array}{ccccccc} TA & \equiv & \widehat{T} * A & \xrightarrow{\text{lift} G_A^{\widehat{T}}} & R[\widehat{T}](H*)A & \xrightarrow{(R[\widehat{T}](H*\leq\top))_A} & R[\widehat{T}]\top A \equiv R_u[T]A \\ Tf \downarrow & & \widehat{T}*f \downarrow & & R[\widehat{T}](H*)f \downarrow & & R[\widehat{T}]\top f \downarrow \\ TB & \equiv & \widehat{T} * B & \xrightarrow{\text{lift} G_B^{\widehat{T}}} & R[\widehat{T}](H*)B & \xrightarrow{(R[\widehat{T}](H*\leq\top))_B} & R[\widehat{T}]\top B \equiv R_u[T]A \\ & & & & & & \downarrow R_u[T]f \end{array}$$

□

We can interpret this relationship in the similar way to the analogous relationship between monads and analysis-graded monads: every analysis-graded monad has a corresponding monad that erases all of the analysis information, and likewise every graded monad transformer also has a corresponding monad transformer obtained by erasing analysis information.

Distributivity and graded monad transformers

The definition of graded monad transformers is very general and imposes few conditions on how a graded monad is mapped to a new one. It is also not the only notion used to talk about combining graded monads. One of them, *distributing graded monads*, has some intersection with graded monad transformers – a relationship we now explore.

In category theory, for two monads $\mathbf{S} = (S, \mu^S, \eta^S)$ and $\mathbf{T} = (T, \mu^T, \eta^T)$, there is no natural monad structure on the functor ST . However, if $\mathbf{S}$ distributes over $\mathbf{T}$, then such a monad exists. In the context of monads, a distributive law of $\mathbf{S}$ over $\mathbf{T}$ is a natural transformation $\theta : TS \Rightarrow ST$ that respects multiplication and unit for $\mathbf{S}$ and $\mathbf{T}$.

In the graded setting, things are somewhat more difficult. Given two graded monads, taking the direct product of their grading algebras as the grading algebra of their composition is overly specific. We adapt a concept from group theory which generalises this product:

Definition 7.5. Let $\mathbb{E}_1 = (E_1, \leq_1, \odot_1, \mathbf{1}_1)$ and $\mathbb{E}_2 = (E_2, \leq_2, \odot_2, \mathbf{1}_2)$ be two computational grading algebras, and let $\alpha : E_2 \times E_1 \rightarrow E_1$ and $\beta : E_2 \times E_1 \rightarrow E_2$ be two monotonic mappings satisfying:

$$\begin{array}{ll} \alpha(\mathbf{1}_2, e_1) &= e_1 & \alpha(e_2 \odot_2 e'_2, e_1) &= \alpha(e_2, \alpha(e'_2, e_1)) \\ \alpha(e_2, \mathbf{1}_1) &= \mathbf{1}_1 & \alpha(e_2, e_1 \odot_1 e'_1) &= \alpha(e_2, e_1) \odot_1 \alpha(\beta(e_2, e_1), e'_1) \\ \beta(\mathbf{1}_2, e_1) &= \mathbf{1}_2 & \beta(e_2, e_1 \odot_1 e'_1) &= \beta(\beta(e_2, e_1), e'_1) \\ \beta(e_2, \mathbf{1}_1) &= e_2 & \beta(e_2 \odot_1 e'_2, e_1) &= \beta(e_2, \alpha(e'_2, e_1)) \odot_2 \beta(e'_2, e_1) \end{array}$$

Then the *Zappa-Szép product* [Uus18] for these grading algebras and mappings $\mathbb{E}_1 \otimes_{\alpha, \beta} \mathbb{E}_2$ is the computational grading algebra with carrier set $E_1 \times E_2$, partial order $\leq_1 \times \leq_2$, unit $\langle \mathbf{1}_1, \mathbf{1}_2 \rangle$ and multiplication defined as follows:

$$\langle e_1, e_2 \rangle \odot \langle e'_1, e'_2 \rangle = \langle e_1 \odot_1 \alpha(e_2, e'_1), \beta(e_2, e'_1) \odot_2 e'_2 \rangle$$

It is not difficult to show that the resulting product $\mathbb{E}_1 \otimes_{\alpha, \beta} \mathbb{E}_2$ is a computational grading algebra (monotonicity of $\odot$, existence of a top element and of joins).

For a distributivity law of analysis-graded monads, we make the grading algebra of the combined graded monad $\mathbb{E}_1 \otimes_{\alpha, \beta} \mathbb{E}_2$, and the *distributivity law* in this case is parameterised by the grades for the two graded monads.

Definition 7.6. For two analysis-graded monads:

$$\begin{aligned} \mathbf{T}_1 &= (T_1, \mu_1, \eta_1) : (E_1, \leq_1, \odot_1, \mathbf{1}_1) \rightarrow \mathbf{Endo}(\mathbb{C}) \\ \mathbf{T}_2 &= (T_2, \mu_2, \eta_2) : (E_2, \leq_2, \odot_2, \mathbf{1}_2) \rightarrow \mathbf{Endo}(\mathbb{C}) \end{aligned}$$

and a Zappa-Szép product of their grading algebras with mappings α and β , a *distributive law* is a family of natural transformations $\theta_{e_2, e_1} : T_2 e_2 \bullet T_1 e_1 \Rightarrow T_1(\alpha(e_2, e_1)) \bullet T_2(\beta(e_2, e_1))$ for $e_1 \in E_1$ and $e_2 \in E_2$ that respects the units and multiplication, meaning that the diagrams in Figure 7.1, as well as the following two, commute:

$$\begin{array}{ccc} T_1 e_1 & \xrightarrow{T_1 e_1 \bullet \eta_2} & T_1 e_1 \bullet T_2 \mathbf{1}_2 \\ \eta_2 \bullet T_1 e_1 \downarrow & \searrow & \\ T_2 \mathbf{1}_2 \bullet T_1 e_1 & \xrightarrow{\theta_{\mathbf{1}_2, e_1}} & T_1 e_1 \bullet T_2 \mathbf{1}_2 \end{array} \quad \begin{array}{ccc} T_2 e_2 & \xrightarrow{\eta_1 \bullet T_2 e_2} & T_1 \mathbf{1}_1 \bullet T_2 e_2 \\ T_2 e_2 \bullet \eta_1 \downarrow & \searrow & \\ T_2 e_2 \bullet T_1 \mathbf{1}_1 & \xrightarrow{\theta_{e_2, \mathbf{1}_1}} & T_1 \mathbf{1}_1 \bullet T_2 e_2 \end{array}$$

Theorem 7.3 ([Uus18]). Let $\mathbf{T}_1 = (T_1, \mu_1, \eta_1) : (E_1, \leq_1, \odot_1, \mathbf{1}_1) \rightarrow \mathbf{Endo}(\mathbb{C})$ and $\mathbf{T}_2 = (T_2, \mu_2, \eta_2) : (E_2, \leq_2, \odot_2, \mathbf{1}_2) \rightarrow \mathbf{Endo}(\mathbb{C})$ be two analysis-graded monads with distributive law $\theta_{e_2, e_1} : T_2 e_2 \bullet T_1 e_1 \Rightarrow T_1(\alpha(e_2, e_1)) \bullet T_2(\beta(e_2, e_1))$. Then the following is a graded monad:

$$\begin{aligned} T\langle e_1, e_2 \rangle &= T_1 e_1 \bullet T_2 e_2 \\ \mu_{\langle e_1, e_2 \rangle, \langle e'_1, e'_2 \rangle} &= ((\mu_1)_{e_1, \alpha(e_2, e'_1)} \bullet (\mu_2)_{\beta(e_2, e'_1), e'_2}) \circ (T_1 e_1 \bullet \theta_{e_2, e'_1} \bullet T_2 e'_2) \\ \eta &= \eta_1 \circ \eta_2 \end{aligned}$$

This construction can be used to, for example, combine any graded monad with **GMaybe** – which is the construction we saw in **GMaybeT**. We now state a more general result and connection.

Graded monad composition as a way of defining a graded monad transformer linked to a particular analysis-graded monad requires us to define a distributivity law with any other analysis-graded monad – composition of two graded monads does not give a monadic structure by default. This is, in fact, the statement of the following theorem:

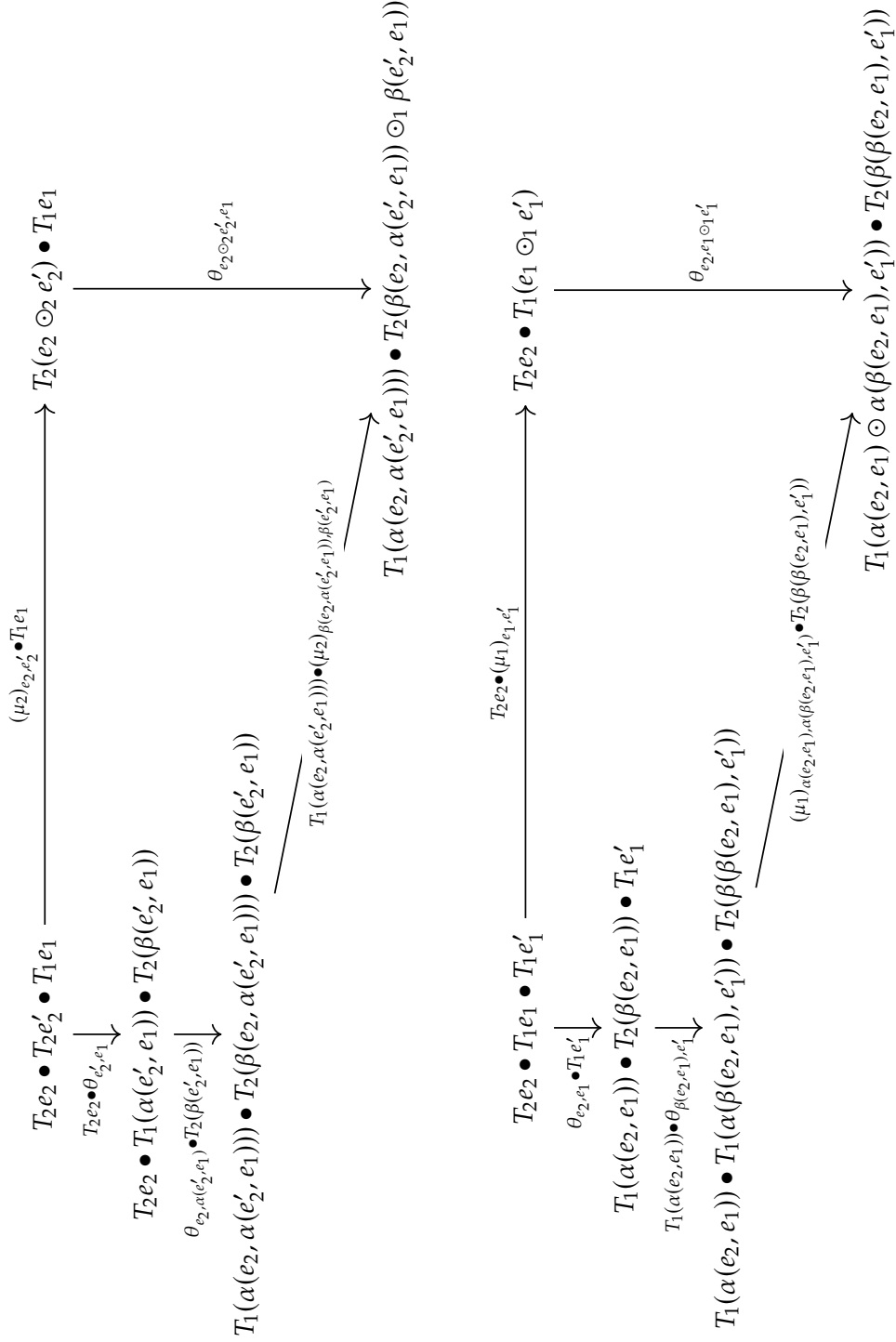


Figure 7.1. Graded distributive law properties: respecting multiplication.

Theorem 7.4. Let $\mathbf{T} = (T, \mu, \eta) : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$ be an analysis-graded monad and let $\theta^{\widehat{\mathbf{T}}}$ be a family of distributivity laws that hold for $\mathbf{T}$ over $\widehat{\mathbf{T}}$ for all analysis-graded monads $\widehat{\mathbf{T}}$ (with mappings $\alpha_{\widehat{\mathbf{T}}}$ and $\beta_{\widehat{\mathbf{T}}}$). Then (R, H, liftG) is an analysis-preserving graded monad transformer, where:

$$\begin{aligned} R[\widehat{T}]\langle e, \widehat{e} \rangle &= Te \bullet \widehat{T}\widehat{e} \\ R[\widehat{T}](\langle e, \widehat{e} \rangle \leq \langle e', \widehat{e}' \rangle) &= T(e \leq e') \bullet \widehat{T}(\widehat{e} \leq \widehat{e}') \\ (R[\widehat{\mu}])_{\langle e, \widehat{e} \rangle, \langle e', \widehat{e}' \rangle} &= (\mu_{e, \alpha_{\widehat{\mathbf{T}}}(\widehat{e}, e')} \bullet \widehat{\mu}_{\beta_{\widehat{\mathbf{T}}}(\widehat{e}, e'), \widehat{e}'}) \circ (Te \bullet \theta_{\widehat{e}, e'} \bullet \widehat{T}\widehat{e}') \\ R[\widehat{\eta}] &= \eta \circ \widehat{\eta} \\ H^{\widehat{\mathbf{T}}}\widehat{e} &= \langle \mathbf{1}, \widehat{e} \rangle \\ \text{liftG}_{\widehat{e}}^{\widehat{\mathbf{T}}} &= \eta_{\widehat{T}\widehat{e}} \end{aligned}$$

Likewise, if $\theta^{\widehat{\mathbf{T}}}$ is a family of distributivity laws that hold for $\widehat{\mathbf{T}}$ over $\mathbf{T}$ for all analysis-graded monads $\widehat{\mathbf{T}}$ (with mappings $\alpha_{\widehat{\mathbf{T}}}$ and $\beta_{\widehat{\mathbf{T}}}$), there is an analysis-preserving graded monad transformer (R, H, liftG) defined analogously to the above, expect that $R[\widehat{T}]\langle e, \widehat{e} \rangle = \widehat{T}\widehat{e} \bullet Te$.

Proof. The two cases are analogous, so we focus on the former. $R\mathbf{T}$ is a graded monad, as shown by Theorem 7.3. $H^{\widehat{\mathbf{T}}}$ is trivially monotonic. Finally, $\text{liftG}^{\widehat{\mathbf{T}}}$ is natural because for each $\widehat{e}$, $\eta_{\widehat{T}\widehat{e}}$ is natural. $\square$

7.1.4 Semantics of lifting

The categorical machinery of graded monad transformers now allows us to define the semantics of graded monad transformers in a programming setting. We thus consider a variant of the Source language with support for graded monad transformers, which requires only a few additions.

We start by considering the syntactic additions. Firstly, we introduce graded monad transformers R , which in the simplest case all range over some fixed class of ‘graded monad transformer constants’ $\mathcal{R}$. Next, graded monads become a more sophisticated syntactic category T , which range over ‘base’ graded monads $\mathbf{T}$ and applications of graded monad transformers to other graded monads. To support the existence of multiple graded monads, the `return` M construct requires the addition of a subscript clarifying what the graded monad of discourse is. Finally, we add a graded lifting construct, which requires explicit annotations clarifying which graded monad and graded monad transformer are being used.

$$\begin{aligned} R &::= \mathcal{R} \\ T &::= \mathbf{T} \mid R[T] \\ M &::= \dots \mid \text{return}_T M \mid \text{liftG}_T^R M \end{aligned}$$

The old value-sorted terms now have the following typing rules:

$$\begin{aligned} (\text{RET}) \quad \frac{\Gamma \vdash M : A \quad \vdash_{\text{gr}} i = \mathbf{1}_T}{\Gamma \vdash \text{return}_T M : T^i A} \quad (\text{LIFTG}) \quad \frac{\Gamma \vdash M : T^e A \quad \vdash_{\text{gr}} f = H_{\mathbf{T}}^R(e)}{\Gamma \vdash \text{liftG}_T^R M : (R[T])^f A} \end{aligned}$$

The semantics of graded monad transformers can be described in terms of $\llbracket \cdot \rrbracket_{\text{gmt}}$. We again somewhat abuse notation and adopt the convention $\llbracket R \rrbracket_{\text{gmt}} =$

$(R, H^R, \text{lift}G^R)$, $\llbracket T \rrbracket_{\text{gm}} = \mathbf{T} = (T, \mu^T, \eta^T, t^T) : \mathcal{E}^T \rightarrow \mathbf{Endo}(\mathbb{C})$ for simplicity (recall all graded monads are assumed to be strong when defining the categorical semantics). We require an additional definition of the semantics of application of graded monad transformers:

$$\llbracket R[T] \rrbracket_{\text{gm}} = \llbracket R \rrbracket_{\text{gmt}} \mathbf{T} = (\llbracket R \rrbracket_{\text{gmt}}[T], \llbracket R \rrbracket_{\text{gmt}}[\mu^T], \llbracket R \rrbracket_{\text{gmt}}[\eta^T], \llbracket R \rrbracket_{\text{gmt}}[t^T])$$

Otherwise, the only modification to the categorical semantics of Source (previously given in Section 5.2) are the semantics of the new return and graded lifting:

$$\begin{aligned} \llbracket \Gamma \vdash \text{return}_T M : T^i A \rrbracket_{\text{val}} &= \eta_{\llbracket A \rrbracket_{\text{type}}}^T \circ \llbracket \Gamma \vdash M : A \rrbracket_{\text{val}} & (i = \mathbf{1}^T) \\ \llbracket \Gamma \vdash \text{lift}G_T^R M : T^f A \rrbracket_{\text{val}} &= (\text{lift}G_T^R)_{\llbracket A \rrbracket_{\text{type}}} \circ \llbracket \Gamma \vdash M : T^e A \rrbracket_{\text{val}} & (f = H_T^R(e)) \end{aligned}$$

7.2 Grade families

In the introduction to this chapter I mentioned that, in addition to graded monad transformers, another way of composing effects represented by graded monads is by adding an new analysis, but not a new side-effect. Consider a computation that operates on a single mutable state of some type S . We have explored several graded monads that represent such computations, but let us consider PermState_S and HistState_S . There is nothing fundamentally different between the effects and computations represented by $\text{PermState}_S F A$ and $\text{HistState}_S G A$, yet they represent two different analyses and the representation of the types (categorically, how the functors $\text{PermState}_S F$ and $\text{HistState}_S G$ act on objects and morphisms). This means that operations that concern these graded monads, such as getting the current value of the state, have representations and uses that depend on the graded monad. In practical terms, this means that a Haskell representation of these has two separate `getPerm` and `getHist`.¹ Combining these two analyses would mean coming up with a general notion of what it means to ‘get’ something – so it would be possible for the programmer to write `get` and have it refer to the trivially-graded version of `State`, `PermState`, `HistState`, or any combination of these analyses – and only specify which one is needed in one place, elsewhere. This is very similar to the Haskell `MonadState` type class, except in a graded setting and where we are not including other side-effects at the same time.²

This gives us the notion of a *grade family*, representing a set of graded monads which represent the same underlying side-effect, but include different kinds of analyses. In our example, PermState_S , HistState_S , as well as their combination, which we call PermHistState_S and which carries information both about the permission and the intermediate values, all belong to the same *stateful grade family*. A stateful graded monad is a graded monad that represents side-effects represented by `get` and `put` computations.

This idea generalises to any number of graded monads that may form a family. If there are three simple analyses represented by three graded monads $\mathbf{T}_1$, $\mathbf{T}_2$ and $\mathbf{T}_3$,

¹This is not unlike ask of the Reader monad differing from `get` associated with `State`.

²One thing omitted from the presentation of graded monad transformers in this chapter is indeed the fact that applying any graded monad transformer to some graded state monad results in a graded monad with a concept of state. In other words, it is useful to show that the derived graded monad also has applicative ‘get’ and ‘put’ operations. For the sake of simplicity, this discussion is omitted from the thesis, but programming-wise it is a matter of defining proper type class instances, which can vary in difficulty and length of implementation.

and their grading algebras are respectively $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{E}_3$, and all of them represent the same underlying effect, then they form an grade family, and furthermore their combinations (any two, all three) should also belong to this grade family. Given that the result of an analysis of a computation should be the same no matter whether it is performed in conjunction with or in isolation from another one, the grading algebras associated with a graded monad combining two analyses is the direct product of algebras representing these analyses.³

We first look at a way that the example of a stateful grade family can be embedded into Haskell. Afterwards, we give a brief and simple theoretical treatment of the concept. Finally, we go back to the programming view, but this time in order to give a semantics for a variant of Source with graded lifting.

7.2.1 Haskell embedding

Each grade family can be represented as a type class in Haskell, specifying exactly the operations one would expect from each graded monad that belongs to it. For example, the stateful grade family described earlier would be represented in the following way:

```
class GMonad m => StateFam (s :: *) (m :: k -> * -> *) where
  type RdGrade m :: k
  type WrGrade m :: k
  get :: m (RdGrade m) s
  put :: s -> m (WrGrade m) ()
```

This definition states that a graded monad belonging to this stateful grade family must have grades associated with reading and writing computations, as well as come with implementations of two functions: `get` and `put`. This means that the type of `get` is very general:

```
get :: StateFam s m => m (RdGrade m) s
```

Two of the interesting graded monads we have dealt with thus far are members of `StateFam`, and these are the instance definitions:

```
instance StateFam s (PermState s) where
  type RdGrade (PermState s) = R0
  type WrGrade (PermState s) = W0
  get = getPerm
  put = putPerm
instance StateFam s (HistState s) where
  type RdGrade (HistState s) = Z
  type WrGrade (HistState s) = S Z
  get = getHist
  put = putHist
```

In addition, there is a graded monad combining these two effects `PermHistState s` that also belongs to this grade family. There is no obvious general way of constructing a representation that for any two graded monads in the same grade family creates a third one combining them, so it is up to the programmer to define all possible combinations. In the interest of brevity, we omit its (lengthy) definition. Firstly, we note that it is an instance of the `StateFam s` type class, easily:

³Of course, a read-only stateful computation necessarily does not have any interesting intermediate history when it comes to the values of the state. We ignore such interactions for simplicity.

```

instance StateFam s (PermHistState s) where
  type RdGrade (PermHistState s) = '(R0, Z)
  type WrGrade (PermHistState s) = '(W0, S Z)
  get = getPermHist
  put = putPermState

```

Furthermore, we want to show that each of the two analyses are subsumed by the analysis given by `PermHistState` – essentially saying that it should be possible to omit the result of one of the analyses. This can also be seen as ‘analysis-downcasting’. This is the essence of graded monads that combine two analyses. To this end we add a new type class, `AnalysisCombination`, which takes three graded monads as parameters, stating that the third graded monad is a combination of the first two:

```

class (GMonad (m :: k -> * -> *), GMonad (n :: l -> * -> *),
        GMonad (p :: (k, l) -> * -> *))
=> AnalysisCombination m n p | m n -> p where
  projFst :: p '(e, f) a -> p e a
  projSnd :: p '(e, f) a -> p f a

```

It is possible, but not essential, to create an `EffFamily` type class that represents grade families, and then impose that `m`, `n` and `p` all belong to the same grade family.

In principle, it is not difficult to embed grade families into Haskell. In practice, however, clear problems arise already when attempting to implement `PermHistState`: the derived graded bind operation requires taking care of all of the possible cases and interactions (thankfully simplified by the fact that it is impossible for an expression to have grades `'(AW, Z)` and `'(R0, S (S Z))`). This gets amplified when there are n graded monads that are used to build an grade family: there are $2^n - 1$ members of this grade family, so we need to define $2^n - n - 1$ additional graded monads. Large grade families thus quickly become impractical, especially when the analyses in question are sophisticated.

7.2.2 Formal definition

The `AnalysisCombination` type class defined above motivates a way of defining grade families more formally. We formally define the notion of *composite graded monads*, informally corresponding to our notion of combination of analyses.

Definition 7.7. Let $\mathbf{S} : \mathbb{E} \rightarrow \mathbf{Endo}(\mathbb{C})$ and $\mathbf{T} : \mathbb{F} \rightarrow \mathbf{Endo}(\mathbb{C})$ be two analysis-graded monads. Denote by $\mathbb{E} \otimes \mathbb{F}$ their product monoidal category. Suppose $\mathbf{U} : \mathbb{E} \otimes \mathbb{F} \rightarrow \mathbf{Endo}(\mathbb{C})$ is also a graded monad. Let $\phi^{\mathbf{S}}$ and $\phi^{\mathbf{T}}$ be families of natural transformations $\phi_{\langle e, f \rangle}^{\mathbf{S}} : \mathbf{U}\langle e, f \rangle \Rightarrow \mathbf{S}e$ and $\phi_{\langle e, f \rangle}^{\mathbf{U}} : \mathbf{U}\langle e, f \rangle \Rightarrow \phi^{\mathbf{T}}f$. We then call $\mathbf{U}$ the *composite graded monad* over $\mathbf{S}$ and $\mathbf{T}$, and we call $\mathbf{S}$ and $\mathbf{T}$ the *components* of $\mathbf{U}$.

Once there is a notion of combining two grading algebras (meaning representing two analyses), defining an grade family is easy.

Definition 7.8. For a number $n \geq 1$, an *grade family* is the set of graded monads T_U for non-empty sets $U \subseteq [n] = \{1, \dots, n\}$ such that for all $U = \{k\} \cup (U \setminus \{k\})$ with $U \setminus \{k\} \neq \emptyset$, and $k = \min U$, the graded monad T_U is a combined graded monad over $T_{\{k\}}$ and $T_{U \setminus \{k\}}$.

7.3 Discussion and further work

This chapter considers two ways of combining analyses in a graded monadic setting: graded monad transformers and grade families. However, the theory and implementation developed in the chapter are merely an introduction, and they give rise to interesting questions which are beyond the scope of this thesis. I briefly discuss possible extensions of this work.

Generalising the distributivity connection

The statement of Theorem 7.4 gives a way of characterising some graded monad transformers as graded monad composition and relates it to the idea of distributive laws. An example which shows how this theorem can be applied is the `GMaybeT` graded monad transformer. However, a more general result would be one that considers more involved ways of composing functors, such as the ones seen in `PermStateTA` and `HistStateTA` graded monad transformers.

A promising result would be one inspired by the idea of transporting structures along adjoint functors. It was first shown by Street [Str72b], given two adjoint functors $U \dashv V, U : \mathbb{C} \rightarrow \mathbb{D}, V : \mathbb{D} \rightarrow \mathbb{C}$, then for any monad $\mathbf{T} = (T, \mu, \eta)$ on category $\mathbb{D}$, there is a monadic structure associated with the functor $VTU : \mathbb{C} \rightarrow \mathbb{C}$. In particular, when $\mathbb{C} = \mathbb{D}$, the two functors U and V give rise to a monad transformer.

More generally, we are looking for a similar criterion in a graded setting: given two functors $U, V : \mathbb{E} \rightarrow \mathbf{Endo}(\mathbb{C})$, we are interested in the conditions under which for any graded monad $\mathbf{T} : \mathbb{F} \rightarrow \mathbf{Endo}(\mathbb{C})$ there is a graded monadic structure associated with $V[T]\langle e, f \rangle = Ve \bullet Tf \bullet Ue$. Such a result would generalise both the distributivity connection made in Theorem 7.4 and the adjunction condition seen for monads.

A different formal treatment of grade families

Definitions 7.7 and 7.8 state an important property of grade families – namely, that analysis information can be erased from a composite graded monad to get a ‘simplified’ member of an grade family. There is no deeper semantic condition: there is nothing that says that all ‘get’ operations are necessarily the same. This would require amending the definition of grade families.

In the context of stateful computations, we would want to show that all ‘get’ operations are the same. Each graded monad $\mathbf{T}_S$ that is part of this grade family has in its grading algebra a member $\text{rd}_{\mathbf{T}_S}$, and there is an associated morphism $\text{get}^{\mathbf{T}} : 1 \rightarrow T_S \text{rd}_{\mathbf{T}_S} S$. ‘Forgetting’ a part of the analysis associated with a get should give exactly the get of the remaining graded monad – for example:

$$\text{get}^{\text{PermState}_S} = \phi_{\langle \text{rd}_{\text{PermState}_S}, \text{rd}_{\text{HistState}_S} \rangle}^{\text{PermState}_S} \circ \text{get}^{\text{PermHistState}_S}$$

Such a correspondence is more complicated to state for put, as these morphisms for each graded monad $\mathbf{T}$ that is part of this grade family would be $\text{put}^{\mathbf{T}} : 1 \rightarrow (T_S \text{wr}_{\mathbf{T}} 1)^S$.

Graded monad transformer families

Both grade families and augmenting graded monad transformers take the view that analyses can be combined in such a way that the new graded algebra has as its carrier set the product of the carrier sets of the original algebras. Other than that, their

roles differ and they can coexist within the same programming language (indeed, the Haskell embedding shows that this is not too difficult). The main practical problem is the implementation length, to which there is no obvious solution.

However, just as we can come up with an grade family, we can also consider the idea of a *graded monad transformer family*: combining the work done by two related graded monad transformers with the same underlying effect into one graded monad transformer that uses both analyses. There is nothing fundamentally different about the side-effect added to a computation by `HistStateTS` and `PermStateTS`, but their analyses differ. Therefore, it makes sense to consider a graded monad transformer that adds *both* of the analyses associated with them, and call it `PermHistStateTS`. This would require defining a notion of composite graded monad transformers – which in practice means implementing all of the $2^n - 1 - n$ ways that n analysis-graded monads that form the foundation of an grade family can interact with each other.

Related notions of combining effects

The term ‘graded monad transformer’ does not make its first appearance in this thesis. The only prior appearance of this term seems to be Julian Grove’s PhD thesis in linguistics [Gro19], where graded monads are seen as a tool for analysing *linguistic side-effects*. [Sha05] This is a general term referring to noncompositional phenomena in natural languages such as anaphora resolution,⁴ intensionality and quantification, which respectively correspond to state, read-only environments and delimited control in programming. Within this context, Grove introduces several graded monad transformers, and the key difference is that the grading algebra always remains fixed.

Kammar in his PhD thesis [Kam14] investigates the question of combining effects in the context of an effectful language with *call-by-push-value* (CBPV) semantics, used as the setting for exploring program optimisations in a Lucassen&Gifford-style type-and-effect system. The language is impure (it features such as call-with-current-continuation and mutable stores), and Chapter 12 of the thesis analyses the conditions under which different side-effects interact with each other. This informs an optimisation procedure of which program transformations are safe. The work done in this area is very complicated and shows how reasoning about effect interaction in an impure language is difficult, and especially how it affects program analysis.

7.4 Summary

In this chapter, I introduced two main ways of combining graded monads and the analyses associated with them. The first, graded monad transformers, provide a way of adding a new side-effect and associated analysis to a graded monadic computation. The second, grade families, are a way of organising analysis-graded monads based on the same side-effect, but provide different analyses.

The most applicable graded monad transformers – analysis-preserving graded monad transformers – can be embedded into existing pure functional programming

⁴Anaphora refers to word repetition, and anaphora resolution refers to reconstruction such repeated words based on pronouns or other substitute words. For example, in “Alan is my supervisor. He is very helpful”, the word ‘he’ refers to ‘Alan’ from the antecedent sentence. In that sense, this is a noncompositional phenomenon.

languages without much difficulty and with an amount of coding effort comparable to that of analysis-graded monads. Graded monad transformers have a simple categorical view and generalise the notion of monad transformers (as shown by Theorem 7.1).

Grade families, by contrast, require the programmer writing the graded monad library to manually predict every possible interaction between related graded monads, meaning an exponential amount of effort required for every new graded monad that is part of an existing grade family. However, when this work is complete, the programmer is able to use operations associated with this grade family without specifying in every place which analyses interest them.

Part IV

Concluding remarks

This part of the thesis consists of two short chapters.

Chapter 8 combines insight from previous chapters to suggest language features to make analysis-graded monads usable. Finally, Chapter 9 summarizes the thesis and outlines further work.

In this thesis, the idea of programming with graded monads has been described in terms of a variety of programming languages. Ideas such as graded monad builders and the semantics of analysis-graded monads have been described using simple and limited languages and calculi such as *Source* and *ML_G*. Haskell is used as a convenient, widely used language in which some of these concepts can be embedded (using the *analysis-monad* library). However, this embedding has its limits, and I argue that Haskell (and other existing languages) are inadequate for practical programming with analysis-graded monads.

In this chapter, I propose a language called *Granalyse*, which aims to provide the features necessary for practical programming with analysis-graded monads. This means supporting ideas such as graded monad builders, and in particular allowing the programmer to define their own grading algebras in a convenient way. The description of the language is informal – I do not delve into the details of its syntax. This chapter can be read as an extended ‘further work’ part of the conclusion of this thesis.

Chapter structure In Section 8.1 I first outline the limitations that exist in Haskell and related languages that prevent all of the interesting features from being implemented. These problems inform the proposed features of *Granalyse*, which are described in Section 8.2.

8.1 Present limitations and difficulties

This section addresses the limitations and other ways in which Haskell is inadequate for programming with analysis-graded monads. These downsides are not unique to Haskell and similar languages – variants of these issues hold for dependently-typed proof assistant languages (such as *Agda* and *Coq*) as well.

Defining grading algebras

The go-to approach to defining grading algebras is by representing grades as types, and the associated operators (sequencing, members of *O*) are all type operators or represented using type families. In some languages, a separate kind can be defined to represent carrier sets of the grading algebra (in Haskell, this is achieved using the *DataKind* extension).

The downside of this approach is that existing facilities for type-level programming require operators to be total. This decision is seemingly sensible: allowing type-level computation to be partial/non-terminating leads to non-termination of

type-checking. This is undesirable: in programming we see typing as a finite, terminating process. The way totality of type-level operators is enforced is by, for example, requiring that type operators are primitive recursive. More sophisticated methods for ensuring termination of type families exist – see the treatment by Eisenberg et al. [EVPJW14] for a discussion on how to infer type family termination. While we want grade operators to be total operators (as is required of analysis-graded monads), insisting on making all functions primitive recursive can be a big burden on the programmer.

Graded monad builders

In Chapter 6, we considered graded monad builders as a syntactic construct that generates a definition of a grading algebra – along with all the necessary operators – as well as the definition of the graded monadic bind, return, subeffecting and the fundamental side-effecting operations (such as stateful reads and writes). These are difficult to embed into existing programming languages, as they require ways of generating type-level code operating on grading algebras parameterised by a set of symbols, as well as the definitions of graded monads. In principle, all of these ideas can be embedded into a language if it has metaprogramming support. For example, lenses in Haskell rely on TemplateHaskell metaprogramming facilities to represent functional references.

Writing a generic graded monad builder should not involve a significant amount of complicated metaprogramming that is difficult to reason about.

Lack of type inference help

Type and grade inference was considered in Chapter 4 where we considered two standard extensions to type inference: recursion and polymorphic grades. Embedding these into existing languages is difficult, especially recursion.

The type inference algorithms of Haskell and other languages do not come with an equation solving procedure. Furthermore, the type system of Haskell does not permit type-level recursion (the types are not *equirecursive*, but *isorecursive*). Ideally, practical programming with grades should have a facility for specifying how recursive grade equations are solved.

Similarly, polymorphism requires equational reasoning with variables – for example, the type system needs to be aware of the fact that for any natural number n , $n + 1$ and $1 + n$ are equal natural numbers. This is something that is not difficult to achieve when all typing is explicit: this is done all the time in dependently-typed languages, and Haskell can simulate it using type equality constraints (for example, $\text{Add } n \ (S \ Z) \sim \text{Add } (S \ z) \ n$, if $S \ Z$ is the type-level representation of 1 and Add is the type family representing addition). While the Haskell type system is capable of inferring which such constraints are ‘missing’, meaning determining which assumptions are necessary for a term to be typeable, it is not capable of automatically showing that these type equalities hold for any valid grade – the range of possible values for type/grade variables such as n is unknown.

8.2 The Granalyse language

All of the problems listed in Section 8.1 are fundamentally a consequence of the kind of grade not being given special treatment in any language. I propose that

a language with support for representing effect systems and program analyses involving side-effecting computation needs to have special capabilities related to defining program analyses and grading algebras. Otherwise, Granalyse would be like any other pure functional language.

Granalyse makes exactly this decision: grades get special treatment. In short, it can be seen as *two* languages: a pure typed functional Core language similar to Haskell used for writing programs and computations, and a more restricted Grade language whose role is only to define grading algebras. While the Core language is able to refer to terms of a grade kind, Grade defines the grading algebras grades may belong to. We explore the features expected of Core and Grade in the remainder of this section.

In Section 8.2.1 we more precisely establish the roles of Core and Grade, including questions about how grade inference is handled by the language. Then we consider the question of how recursion would be supported in this Section 8.2.2. Finally, support for graded monad builders and symbols is explored in Section 8.2.3.

8.2.1 A two-level language

Core is a language with the usual features of a pure functional language with some useful extensions: call-by-name semantics, lazy evaluation, algebraic data-types and *generalised* algebraic data-types (GADTs), basic type class support – including type classes with multiple parameters. These features allow Core to define structures such as monads and comonads. The key addition is the fact that the Core language identifies special kinds, those being *grades from a particular grading algebra*. Type constructors and multiple-parameter type classes are able to take grade-kinded parameters, introducing language support for kindanalysis-graded structures – graded monads, graded comonads, or novel and interesting data types. Graded monads, for example, can be represented in a similar way to analysis-graded monads in my `analysis-monad` library – a graded monad is a type class with a type constructor taking a grading algebra-kinded argument and a type-kinded argument, along with binding, return and subeffecting operations. The Core language is parametrically polymorphic in types and grades.

Grade is a less sophisticated functional language which can be seen as defining modules with a very specific use. It has support for algebraic data-types in order to represent numbers, lists and trees, and a very basic ‘type’ system. Each module written in Grade defines a separate grading algebra. There are special definitions that need to be provided in each module: a data type representing the carrier set of the algebra, functions representing the sequencing operator, partial order and the join, grades representing the unit and top elements, and a set of functions representing grade-level operators. We could make the concept more general and view almost all parts of the definition as instances of grade-level operators, introduce a more sophisticated hierarchies (‘grading algebra classes’, one of which would be ‘algebras with a unit’), but we keep the treatment simple and focussed on representing computational grading algebras.

The key limitation of the Haskell embedding is the requirement that grade-level operators need to pass a totality check. Grade gets around this by having no such checks – instead, the default assumption is that grade-level operators are all implemented correctly. This comes at the cost of making type checking Granalyse programs non-terminating in the case this assumption is false. A practical Granalyse

compiler would then enable a flag to specify a timeout as a way of detecting failure of typing.

In addition to implementating the core parts of the grading algebra definitions, the Grade language modules enable the programmer to define useful properties which aid type checking and inference. The universal properties – such as monotonicity – are present by default, but those characteristic for a particular algebra – such as commutativity or idempotence of $\odot$ – can be added. Whether these properties hold is not verified by the compiler, the programmer is taken at their word that the property holds.¹

Constructing grading algebras Graded monad transformers, as described in Section 7.1, require a way of transforming a grading algebra into a graded algebra with pairs of grades. This requires a way of writing parameterised Grade modules: given a grading algebra, define a new, derived grading algebra. Every Grade module is parameterised by zero or more other grading algebras, which are used as variables that range over grading algebras. The module can then refer to names exported by the module – for example, the unit element of algebra E is $E.\text{unit}$.

Grade equations With definitions involving conditional branching and recursion, a common problem arises with grade-polymorphic functions: checking whether two polymorphic grade expressions are equal. The approach of Granalyse is to rely on the facts specified in the Grade modules that define the particular grading algebra. Internally, this verification procedure can be implemented in a similar way to the Granule language [OLEI19]: represent the grade equality problem as a problem for an SMT solver or other kind of theorem prover.

8.2.2 Recursion

Recursion is, as expected, a challenging part of designing type and grade inference algorithms, especially in tandem with polymorphism. The approach Granalyse takes must therefore combine the insights from Chapter 4: it should aim to solve equations where there are no grade variables and use unification-based methods in order to compute a principal type scheme.

When a computational grading algebra has finite height, iteration-based approaches always converge to a solution to a system of grade equations if it exists and is unique (which it is if definitions are monotonic, as promised). The finite height property can then be stated in the algebra definition written in Grade. When this is not stated, the programmer is required to specify an algorithm for solving arbitrary systems of such equations in Grade – meaning a function that returns a solution or a special value representing no existence of said solution. This solves the issues associated with infinite-height algebras (such as those with carrier set $\mathbb{N}_\omega$), but this requires additional syntax that can be considered not part of the ‘main’ Granalyse language.

Limited polymorphic recursion, along the lines of Haskell, would be available. Through the use of properties specified for grading algebras (e.g., operator reflexivity), some number of annotations can be removed (compared to Haskell). Otherwise

¹We could alternatively define Grade as a dependently-typed language and require such facts about a grading algebra to be proved, but the goal of Granalyse is to keep the definition of program analyses simple.

it should be seen as a combination of existing semi-algorithms for type inference in languages with polymorphic recursion and using an SMT solver to fill in the gaps that arise from generated grade equations that need to be verified.

8.2.3 Symbols and builders

In the discussion thus far, the questions of graded monad builders and symbols were not considered as Granalyse features. This requires some further non-trivial additions, and the goal is to make the extensions similar to what we see in System G.

Granalyse additionally has a kind of symbols and a kind of symbols, as well as the option of abstracting over both of them and applying them. For the sake of simplicity, Granalyse avoids the awkward question of symbol scoping by making their scope an entire source file. Writing symbol abstractions would, as in System G, require specifying the symbol set over which the symbol is allowed to range.

In order to allow for symbol-parameterised grading algebras, the Grade modules can be parameterised by sets of symbols usable in definitions of the carrier sets of their algebras and can be parameters of the algebra operators. Graded monad builders can then be represented by making Core type constructors and type classes more permissive: they can take as arguments a symbol set a , a grade drawn from a parameterised grading algebra instantiated with the symbol set, and then a type.

8.3 Summary

I summarized the difficulties associated with embedding analysis-graded monads and computational grading algebras into existing functional languages. I then informally proposed the Granalyse language, whose main goal is to allow the programmer a convenient way to both write programs and introduce analyses they are interested in. This is achieved by making Granalyse actually consist of two languages, Core and Grade, with the former being a pure functional language in the vein of Haskell, and the latter being a language used purely for defining grading algebras. I discussed how this idea can represent concepts such as graded monad transformers, graded monad builders and grade-level equations in type inference.

An important aspect of Granalyse is that the *programmer is responsible* for ensuring that all of the assumptions on grading algebras hold, which include monotonicity of sequencing, computability of all operators and partial order decidability. These properties are not automatically verified, determining it is undecidable in Grade.

I argue that giving the programmer power to define their own verification procedures within a language, without any cost seen in the generated final program (all the additional cost is at compile time), is a powerful tool for developing robust software.

Conclusion

9.1 Summary of contributions

The focus of this thesis has been on programming with graded monads whose grades represent program analyses, and the goal has been to explore alternatives to current programming paradigms.

I considered the category-theoretic foundations of analysis-graded monads, most notably in Chapter 5 and parts of Chapter 7, and thus provided a semantic foundation and justification of the work undertaken. Most of the thesis focused on the various practical questions related to programming with existing and proposed facilities used to represent analysis-graded monads. In Chapter 3 I defined computational grading algebras and considered the potential of using analysis-graded monads in various settings (checked exceptions, state permissions); the original contribution is a novel approach to using graded monads going beyond prior systematic attempts (concurrency being one example). The question of grade inference in languages with analysis-graded monads was considered in Chapter 4 and I showed how the grade inference is an extension of type inference, as well as established the conditions that make this task decidable. Chapters 6 and 7 considered ways in which expressing effect analyses through grades can be made practical using facilities such as graded monad builders for representing sophisticated algebras and graded monad transformers for combining effects – both novel ideas. Finally, in Chapter 8 I described a hypothetical Granalyse language, a pure functional language which gives the programmer power to describe their own interesting analyses in a simple way using grading algebras, and I explored language features necessary to overcome the limitations and difficulties in using analysis-graded monads in existing languages.

An additional contribution of the thesis is the `analysis-monad` Haskell package, which builds upon previous efforts of embedding type-and-effect systems into Haskell. To the best of my knowledge, this is the only ‘complete’ implementation of some analyses (for example, state permissions) that have been attempted before.

9.2 Further work

This thesis explores one of the approaches to expressing behavioural information as part of the type system. It is just one part of the story on programming with effects in functional languages, with a variety of alternative approaches in existence, most notably algebraic effects and their handlers. It is also part of the static analysis story, most related to type-and-effect systems research that started with Lucassen and Gifford’s seminal paper [LG88].

With such a vast body of background and related literature, there is significant potential for future work building upon this thesis. This section contains a non-exhaustive list of suggestions for further work.

9.2.1 Other graded structures

The focus in this thesis has been on effectful computations represented by monads. However, other kinds of impure computation and their respective analyses can be represented using similar constructs. For example, coeffectful computation [POM13] is semantically represented by comonads and graded comonads. Graded joinads can also be used to discuss properties related to classical data-flow analyses [PS11, MOP16], with formalism inspired by Honda’s session types [Hon93].

A natural extension of the work undertaken in this thesis is its adaptation into contexts that are not only monadic. In Chapter 8 I hinted at the idea that programming with grades does not force us to only consider analysis-graded monads, and other graded structures should be definable within Granalyse or a similar language. Likewise, all of the theory developed regarding practical programming, such as graded monad transformers and builders, should be extended to cover ideas such as graded comonads – meaning having concepts such as *graded comonad transformers*.

More complex notions of computation Graded comonad transformers are the first immediately useful and interpretable extension, as this idea allows the programmer to combine outcomes of several coeffect analyses. Furthermore, there is potential to further consider how effects and coeffects interact, going beyond the work of Gaboardi et al. [GKO⁺16] which considered effect/coeffect pairs for which interaction was described using a graded distributive law. There is potential to investigate the circumstances under which graded distributive laws exist, allowing the programmer to combine graded monad and comonad transformers in order to represent a variety of effect-like and coeffect-like analyses (and their combinations) using graded transformers.

9.2.2 Further theoretical results

Chapters 5 and 7 featured theoretical foundations of concepts associated with analysis-graded monads. The results stated in these chapters are meant to be immediately interpretable from a programming side. There are questions worth asking (and answering) regarding the semantics and category-theoretic background of combining program analysis with graded monads.

Syntax for analysis abstraction In Section 5.2, I briefly related the ideas of abstraction and concretisation in abstract interpretation to maps between grades of two different grading algebras. The core idea was that two graded monads can form a pair in which one is ‘more concrete’ and another is ‘more abstract’: they essentially represent the same side-effecting computation, but one gives more precise results. However, the idea of one analysis-graded monad being an abstraction of another one warrants more attention from an operational perspective, not just denotational. This would mean introducing the concept of translation of graded monadic values represented with type T_1A to graded monadic values represented with type T_2A

and vice-versa, translating grades in the process and having *compatible* reduction sequences. That was, a more complete *correct by construction* view of writing programs with graded monadic types might be possible.

Hierarchy of graded monad transformers A hierarchy of monoid transformers was given by Jaskelioff and Moggi [JM10a]: other than monoid transformers, we observe covariant, functorial and monoidal monoid transformers, which have further interesting structural properties. Monad transformers can be seen as monoid transformers. I did not consider any similar hierarchy for graded monad transformers in this thesis. Further research in this area should investigate the properties of graded lifting for various graded monad transformer classes.

Safety theorems While the purpose of this effort is to represent program analyses and reason about safety properties, we never established the relation between the semantics of terms and the grades in their types. Further work could attempt to establish a formal connection through safety theorems which link the grade with the safety guarantee that they are purported to represent.

9.2.3 An implementation of Granalyse

Chapter 8 only proposed Granalyse in an informal way: I did not provide an implementation and did not define any semantics. This proposal is really two languages in one: a Grade language for defining grading algebras (in our context, program analyses) and a Core language for more general-purpose programming.

Programming languages capable of reasoning about resource usage or embedding analyses exist, with Granule [OLEI19] a significant recent development – though without using (graded) monads or comonads as structures to program with. However, an implementation of Granalyse would be novel in the sense that the programmer would have a lot more control over the analyses and properties they wish to express.



In this thesis I have developed a theory for programming with graded monads in the context of program analysis, especially useful features that do not presently exist in any programming language. The Haskell embedding, seen in the `analysis-monad` library, addresses only one part of the story – it does not provide an elegant way of embedding the grading algebras, especially their operators. A project that aims to complete and extend the proposals I made regarding Granalyse would be a unique undertaking and a topic for plenty of language design discussion: a language designed around giving programmers safety guarantees, and having the type system support defining their own safety criteria. I hope that the contributions of this thesis provide useful background in designing languages whose primary goal is to write provably correct programs.

Bibliography

- [Atk09] Robert Atkey. Parameterised notions of computation. *J. Funct. Program.*, 19(3–4):335–376, July 2009.
- [BB07a] Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, page 15–26, New York, NY, USA, 2007. Association for Computing Machinery.
- [BB07b] Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '07, page 15–26, New York, NY, USA, 2007. Association for Computing Machinery.
- [Bie95] G. M. Bierman. What is a categorical model of intuitionistic linear logic? In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 78–93, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.
- [BN15] Jan Bracker and Henrik Nilsson. Polymonad programming in haskell. In *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*, IFL '15, New York, NY, USA, 2015. Association for Computing Machinery.
- [BP12] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *CoRR*, abs/1203.1539, 2012.
- [BR01] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free java programs. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, page 56–69, New York, NY, USA, 2001. Association for Computing Machinery.
- [CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, page 238–252, New York, NY, USA, 1977. Association for Computing Machinery.
- [CC92] Patrick Cousot and Radhia Cousot. Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In

- Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming*, pages 269–295, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.
- [Con11] Conor McBride. Kleisli arrows of outrageous fortune, 2011.
- [DF90] Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, page 151–160, New York, NY, USA, 1990. Association for Computing Machinery.
- [Dijnd] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In *Operating Systems Techniques*, pages 72–93. n.d.
- [DLH10] Ugo Dal Lago and Martin Hofmann. Bounded linear logic, revisited. *Logical Methods in Computer Science*, 6(4), Dec 2010.
- [DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, page 207–212, New York, NY, USA, 1982. Association for Computing Machinery.
- [EVPJW14] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, page 671–683, New York, NY, USA, 2014. Association for Computing Machinery.
- [Fel88] Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, page 180–190, New York, NY, USA, 1988. Association for Computing Machinery.
- [Fil94] Andrzej Filinski. Representing Monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 446–457, New York, NY, USA, 1994. ACM.
- [FKLP17] Yannick Forster, Ohad Kammar, Sam Lindley, and Matija Pretnar. On the Expressive Power of User-defined Effects: Effect Handlers, Monadic Reflection, Delimited Control. *Proc. ACM Program. Lang.*, 1(ICFP):13:1–13:29, August 2017.
- [FKM16] Soichiro Fujii, Shin-ya Katsumata, and Paul-André Mellies. Towards a formal theory of graded monads. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 513–530, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.
- [FQ03a] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. *SIGPLAN Not.*, 38(5):338–349, May 2003.
- [FQ03b] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation*, TLDI '03, page 1–12, New York, NY, USA, 2003. Association for Computing Machinery.

- [GKO⁺16] Marco Gaboardi, Shin-ya Katsumata, Dominic Orchard, Flavien Breuvert, and Tarmo Uustalu. Combining effects and coeffects via grading. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 476–489, New York, NY, USA, 2016. ACM.
- [Gor17] Colin S. Gordon. A generic approach to flow-sensitive polymorphic effects. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19–23, 2017, Barcelona, Spain*, volume 74 of *LIPICs*, pages 13:1–13:31. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017.
- [Gor18] Colin S. Gordon. Sequential effect systems with control operators. *CoRR*, abs/1811.12285, 2018.
- [Gor21] Colin S. Gordon. Polymorphic iterable sequential effect systems. *ACM Trans. Program. Lang. Syst.*, 43(1):4:1–4:79, 2021.
- [Gro19] Julian Grove. *Scope-taking and presupposition satisfaction*. PhD thesis, The University of Chicago, 2019.
- [GS14] Dan R. Ghica and Alex I. Smith. Bounded linear types in a resource semiring. In *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, page 331–350, Berlin, Heidelberg, 2014. Springer-Verlag.
- [GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: a modular approach to polynomial-time computability. *Theoretical Computer Science*, 97(1):1–66, 1992.
- [HBG⁺14] Michael Hicks, Gavin Bierman, Nataliya Guts, Daan Leijen, and Nikhil Swamy. Polymonadic programming. *Electronic Proceedings in Theoretical Computer Science*, 153:79–99, Jun 2014.
- [HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence, IJCAI’73*, page 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
- [Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.
- [Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.
- [Hon93] Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR’93*, pages 509–523, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.
- [IM20] Andrej Ivašković and Alan Mycroft. A graded monad for deadlock-free concurrency (functional pearl). In *Proceedings of the 13th ACM SIGPLAN International Symposium on Haskell, Haskell 2020*, page 17–30, New York, NY, USA, 2020. Association for Computing Machinery.

- [IMO20] Andrej Ivašković, Alan Mycroft, and Dominic Orchard. Data-Flow Analyses as Effects and Graded Monads. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:23, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.
- [Jas09a] Mauro Jaskelioff. *Lifting of Operations in Modular Monadic Semantics*. PhD thesis, University of Nottingham, 2009.
- [Jas09b] Mauro Jaskelioff. Modular monad transformers. In Giuseppe Castagna, editor, *European Symposium on Programming*, volume 5502, pages 64–79. Springer, 2009.
- [Jas11] Mauro Jaskelioff. Monatron: an extensible monad transformer library. In *Proceedings of the 20th international conference on Implementation and application of functional languages, IFL’08*, pages 233–248, Berlin, Heidelberg, 2011. Springer-Verlag.
- [JM10a] Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theor. Comput. Sci.*, 411(51–52):4441–4466, December 2010.
- [JM10b] Mauro Jaskelioff and Eugenio Moggi. Monad transformers as monoid transformers. *Theoretical Computer Science*, 411(51–52):4441 – 4466, 2010.
- [Joh97] Peter Johnstone. Cartesian monads on toposes. *Journal of Pure and Applied Algebra*, 116(1):199–220, 1997.
- [Kam14] Ohad Kammar. Algebraic theory of type-and-effect systems. 2014.
- [Kat14] Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’14*, page 633–645, New York, NY, USA, 2014. Association for Computing Machinery.
- [KG21] Finnbar Keating and Michael B. Gale. *Graded Monads and Type-Level Programming for Dependence Analysis*, page 27–40. Association for Computing Machinery, New York, NY, USA, 2021.
- [KJS10] Oleg Kiselyov, Simon Peyton Jones, and Chung-chieh Shan. *Fun with Type Functions*, pages 301–331. Springer London, London, 2010.
- [Koz90] Dexter Kozen. On kleene algebras and closed semirings. In Branislav Rován, editor, *Mathematical Foundations of Computer Science 1990*, pages 26–47, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.
- [KU77] John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Inf.*, 7:305–317, 1977.
- [Laf88] Y. Lafont. The linear abstract machine. *Theoretical Computer Science*, 59(1):157–180, 1988.
- [Law69] F. William Lawvere. Adjointness in foundations. *Dialectica*, 23(3/4):281–296, 1969.

- [LG88] J. M. Lucassen and D. K. Gifford. Polymorphic Effect Systems. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 47–57, New York, NY, USA, 1988. ACM.
- [LHJ95] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 333–343, New York, NY, USA, 1995. Association for Computing Machinery.
- [LP00] Xavier Leroy and François Pessaux. Type-based analysis of uncaught exceptions. *ACM Trans. Program. Lang. Syst.*, 22(2):340–377, March 2000.
- [MAA⁺19] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hrițcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.
- [Mog89] E. Moggi. Computational lambda-calculus and monads. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science*, page 14–23. IEEE Press, 1989.
- [Mog91] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. Selections from 1989 IEEE Symposium on Logic in Computer Science.
- [MOP16] Alan Mycroft, Dominic Orchard, and Tomas Petricek. *Effect Systems Revisited—Control-Flow Algebra and Semantics*, pages 1–32. 01 2016.
- [MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Information and Computation*, 100(1):1–40, 1992.
- [Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. In M. Paul and B. Robinet, editors, *International Symposium on Programming*, pages 217–228, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.
- [NMB06] Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, page 62–73, New York, NY, USA, 2006. Association for Computing Machinery.
- [NN94] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology (extended abstract). In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, page 84–97, New York, NY, USA, 1994. Association for Computing Machinery.
- [NRNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. 01 1999.
- [OL17] Dominic Orchard and Vilem-Benjamin Liepelt. Gram: A linear functional language with graded modal. In *Workshop on Trends in Linear Logic and Applications (TLLA)*, 2017.

- [OLEI19] Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.
- [OM12] Dominic A. Orchard and Alan Mycroft. A notation for comonads. In *IFL*, 2012.
- [OP14] Dominic Orchard and Tomas Petricek. Embedding Effect Systems in Haskell. *SIGPLAN Not.*, 49(12):13–24, September 2014.
- [OPM14] Dominic A. Orchard, Tomaš Petříček, and Alan Mycroft. The semantic marriage of monads and effects. *CoRR*, abs/1401.5391, 2014.
- [Orc14] Dominic Orchard. Programming contextual computations. Technical Report UCAM-CL-TR-854, University of Cambridge, Computer Laboratory, May 2014.
- [ORY01] Peter O’Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In Laurent Fribourg, editor, *Computer Science Logic*, pages 1–19, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.
- [OWE20] Dominic Orchard, Philip Wadler, and Harley Eades. Unifying graded and parameterised monads. *Electronic Proceedings in Theoretical Computer Science*, 317:18–38, May 2020.
- [OY16] Dominic Orchard and Nobuko Yoshida. Effects as sessions, sessions as effects. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL ’16, page 568–581, New York, NY, USA, 2016. Association for Computing Machinery.
- [Pet17] Tomas Petricek. *Context-aware programming languages*. PhD thesis, University of Cambridge, March 2017.
- [Pie05] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. 2005.
- [Pit87] A. M. Pitts. Polymorphism is set theoretic, constructively. In David H. Pitt, Axel Poigné, and David E. Rydeheard, editors, *Category Theory and Computer Science*, pages 12–39, Berlin, Heidelberg, 1987. Springer Berlin Heidelberg.
- [PJ02] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. 05 2002.
- [POM13] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft. Coeffects: Unified Static Analysis of Context-Dependence. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 385–397. Springer, 2013.

- [POM14] Tomaš Petriček, Dominic A. Orchard, and Alan Mycroft. Coeffects: a calculus of context-dependent computation. In Johan Jeuring and Manuel M. T. Chakravarty, editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 123–135. ACM, 2014.
- [Pot98] François Pottier. Type Inference in the Presence of Subtyping: from Theory to Practice. Research Report RR-3483, INRIA, 1998.
- [PP09] Gordon Plotkin and Matija Pretnar. Handlers of Algebraic Effects. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, ESOP '09*, pages 80–94, Berlin, Heidelberg, 2009. Springer-Verlag.
- [PP13] Gordon D. Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, 9(4), 2013.
- [PS11] Tomas Petricek and Don Syme. Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming. In *Proceedings of Practical Aspects of Declarative Languages, PADL 2011*, 2011.
- [RHS95] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61, 1995.
- [Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.
- [See83] Robert A. G. Seely. Hyperdoctrines, natural deduction and the beck condition. *Mathematical Logic Quarterly*, 29(10):505–542, 1983.
- [Sha05] Chung-chieh Shan. *Linguistic side effects*. PhD thesis, 2005.
- [SNN94] Kirsten Lackner Solberg, Hanne Riis Nielson, and Flemming Nielson. Strictness and totality analysis. In Baudouin Le Charlier, editor, *Static Analysis*, pages 408–422, Berlin, Heidelberg, 1994. Springer Berlin Heidelberg.
- [Ste94] Guy L. Steele. Building interpreters by composing monads. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '94*, page 472–492, New York, NY, USA, 1994. Association for Computing Machinery.
- [Str72a] Ross Street. The formal theory of monads. *Journal of Pure and Applied Algebra*, 2(2):149–168, 1972.
- [Str72b] Ross Street. Two constructions on lax functors. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 13(3):217–264, 1972.
- [Swi09] Wouter Swierstra. A hoare logic for the state monad. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors,

- Theorem Proving in Higher Order Logics*, pages 440–451, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [SWS⁺13] Nikhil Swamy, Joel Weinberger, Cole Schlesinger, Juan Chen, and Benjamin Livshits. Verifying higher-order programs with the dijkstra monad. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 387–398, New York, NY, USA, 2013. Association for Computing Machinery.
- [Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285 – 309, 1955.
- [Tat13] Ross Tate. The sequential semantics of producer effect systems. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, page 15–26, New York, NY, USA, 2013. Association for Computing Machinery.
- [TJ94] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. *Information and computation*, 111(2):245–296, 1994.
- [Uus18] Tarmo Uustalu. Grading monads, comonads and distributive laws, 2018.
- [Uus20] Tarmo Uustalu. Graded monads and quantified computational effects, 2020.
- [UV05] Tarmo Uustalu and Varmo Vene. The essence of dataflow programming. pages 135–167, 11 2005.
- [UV08] Tarmo Uustalu and Varmo Vene. Comonadic notions of computation. *Electron. Notes Theor. Comput. Sci.*, 203(5):263–284, June 2008.
- [WA85] William W. Wadge and Edward A. Ashcroft. *LUCID, the Dataflow Programming Language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.
- [WT03] Philip Wadler and Peter Thiemann. The Marriage of Effects and Monads. *ACM Trans. Comput. Logic*, 4(1):1–32, January 2003.

Part V

Appendices

Index of notation

This Appendix gives an overview of the most common meanings of certain symbols throughout this thesis.

A.1 Mathematics

A.1.1 General

$\odot$	monoid multiplication
$\vee$	join
$\wedge$	meet
$\perp$	bottom element of a partial order
$\top$	top element of a partial order
$\text{dom}(f)$	domain of a partial function (for environments such as Γ : all variable names in the domain)
E	monoid carrier set
$\mathbb{E}$	set E equipped with some partial order (usually $\leq$)
$\mathcal{E}$	effect/grading algebra
$\mathbb{N}$	natural numbers (starting from 0)
$\mathcal{O}$	grading algebra operators
$\mathbb{Z}$	integers
α	abstraction function (abstract interpretation)
γ	concretisation function (abstract interpretation)
ω	'infinite-length' (e.g. ' ω -chain')
	element representing 'infinity'

A.1.2 Category theory

$*$	only object in the trivial one-object category
$+$	coproduct
$\times$	product
$\otimes$	tensor product (usually in a monoidal category)
$\bullet$	functor composition
$\circ$	morphism composition
$[f, g]$	for $f : A \rightarrow X, g : B \rightarrow X$, the universal morphism $A + B \rightarrow X$
$\langle f, g \rangle$	for $f : X \rightarrow A, g : X \rightarrow B$, the universal morphism $X \rightarrow A \times B$

$!_A$	terminal mapping $A \rightarrow 1$
0	initial object
1	terminal object
$\mathbb{1}$	one-object, one-morphism category
$\mathbb{2}$	two-object, two-morphism (only identities) category
$\mathbb{A}, \mathbb{B}, \mathbb{C}, \dots$	category
$\mathbb{A}^{\text{op}}$	category opposite to $\mathbb{A}$
$\mathcal{A}, \mathcal{B}, \mathcal{C}, \dots$	monoidal category
Cat	category of small categories
Endo ($\mathbb{C}$)	category of endofunctors $\mathbb{C} \rightarrow \mathbb{C}$
Endo ₂ ($\mathbb{C}$)	a one-object 2-category whose morphisms with 1-cells functors $\mathbb{C} \rightarrow \mathbb{C}$ and 2-cells natural transformations between these functors
Mnd ($\mathbb{C}$)	category of monads on $\mathbb{C}$
Mon ($\mathbb{C}$)	category of monoids on $\mathbb{C}$
GMnd ($\mathbb{C}$)	category of analysis-graded monads (with ‘small’ grading algebra categories) on $\mathbb{C}$
SGMnd ($\mathbb{C}$)	category of strong analysis-graded monads on $\mathbb{C}$
A, B, C	object
A^B	exponential object (elsewhere $[B, A]$ or $B \Rightarrow A$)
$\text{ev}_{A.B}$	‘uncurry’ morphism $A^B \times B \rightarrow A$
F, G, H	functor
f, g, h	morphism
I	unit object in a monoidal category
Id _{$\mathbb{C}$}	identity functor on category $\mathbb{C}$
id	identity natural transformation
id_A	identity morphism for object A
M	monoid object
m	morphism representing the denotation of a term
R	(graded) monad transformer
T	monad or graded monad
$\widehat{\mathbf{T}}$	graded monad corresponding to monad T
T	monadic or graded monadic functor
t	tensorial strength
α	monoidal category associator
α, β	natural transformation
η	unit natural transformation; monoid unit; unit coherence natural transformation for a lax monoidal functor
$\iota_1^{A,B}$	left injection $A \rightarrow A + B$ (definition of coproduct)
$\iota_2^{A,B}$	right injection $B \rightarrow A + B$ (definition of coproduct)
λ	left unitor in monoidal category
λf	curry morphism for f (definition of exponential)
μ	(graded) monadic multiplication; monoid multiplication; associativity coherence natural transformation for a lax monoidal functor
ρ	right unitor in monoidal category

$\pi_1^{A,B}$	left projection $A \times B \rightarrow A$ (definition of product)
$\pi_2^{A,B}$	right projection $A \times B \rightarrow A$ (definition of product)

A.2 Language syntax

Bool	boolean type (inhabited only by true and false)
Int	integer type
Unit	unit type (inhabited only by $\langle \rangle$)
$-$	ignored variable, wildcard
A, B	type
b	boolean variable name
c	constant (of any type)
e, f, g	grade
F, G, H	effect term
$fev(A)$	effect/grade variables occurring freely in A
$ftv(A)$	type variables occurring freely in A
$fv(M)$	value-kinded variables occurring freely in M
$\mathcal{I}$	shorthand for $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \mid \Gamma$
$\mathcal{J}$	shorthand for $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi$
M, N	(value-kinded) term
m	integer variable name
n	integer constant
op	metavariable ranging over binary operators (usually only on integers)
R	set of effect variable names
r, s	grade variable
S	symbol set
	syntactic type in Source [®]
$\mathcal{S}$	environment of symbol-set variables
$T^\circ S$	computational grading algebra of the application of builder T to symbol set S
$[T]S$	syntax referring to computational grading algebra $T^\circ S$
$T^\blacklozenge S$	graded monad (as a type constructor) resulting from applying builder T to S
t	result of translating a term from Source to Target
V	set of type variable names
x	metavariables ranging over program variables
$x, y, z, \dots$	variable names in function abstractions
α, β, γ	type variable
Γ	type environment
Δ	environment of symbol variables (introduced through symbol abstractions)
δ	symbol variable
θ	substitution
Σ	environment of symbols added through new X within ...
σ	type scheme
τ	type without any grades (in ML_G)
Φ	environment of grade variables

ξ	symbol-set variable
Ψ	symbol term

Category-theoretic background

Most contributions of the thesis do not rely on knowledge of category theory. This appendix covers concepts from masters-level courses that are used or mentioned in Chapters 2, 5 and 7: categories, functors, natural transformations, cartesian closed categories (ccc), (co)limits, monoidal categories, and a short explanation of 2-categories (which are occasionally referenced in discussion, but no original contribution relies on knowledge of this concept).

This appendix does not aim to teach the reader category theory and should instead be seen as a quick reference (and possibly review of notation). A detailed account of all the terms used in this appendix can be found in most textbooks on the topic.

Broadly speaking, category theory is a tool for describing and reasoning about *universal properties*, allowing us to find common structure between various related notions – for example, the fundamental idea between the cartesian product of sets and direct product of groups is the same when viewed through a categorical lens.

B.1 Categories

B.1.1 Definition of a category

We start from the standard, classical definition of categories in terms of objects and morphisms:

Definition B.1. A *category* $\mathbb{C}$ is defined by:

- a class of *objects* $\text{Obj}(\mathbb{C})$;
 - for any two objects A, B , a class of *morphisms* (or *maps*) $\text{Hom}_{\mathbb{C}}(A, B)$ (sometimes written $\mathbb{C}(A, B)$) with *source* A and *target* B , denoted $f : A \rightarrow B$ for a specific morphism;
 - for any two morphisms $g : B \rightarrow C$, $f : A \rightarrow B$, a *composition* of these morphisms is a morphism $g \circ f : A \rightarrow C$;
 - for any object A , there is an identity morphism $\text{id}_A : A \rightarrow A$ such that for all morphisms $f : A \rightarrow B$, $f \circ \text{id}_A = f$, and for all $g : B \rightarrow A$, $\text{id}_A \circ g = g$;
 - for any three morphisms $f : A \rightarrow B$, $g : B \rightarrow C$, $h : C \rightarrow D$, the composition operator associates: $(h \circ g) \circ f = h \circ (g \circ f)$.
-

It is sometimes useful to consider categories in which classes $Obj(\mathbb{C})$ and all $Hom_{\mathbb{C}}(A, B)$ are sets, and we call these categories *small*. If all the morphism classes $hom(A, B)$ are sets, then the category is called *locally small*.

For a particular category $\mathbb{C}$, one can construct several kinds of derived categories. The first notion is thinness – constraining the cardinality of the number of morphisms.

Definition B.2. A category is *thin* if for any two objects A and B in it, there is at most one morphism from A to B .

Next, we have categories in which no two objects are ‘essentially the same’:

Definition B.3. A morphism $f : A \rightarrow B$ is an *isomorphism*, and thus objects A and B are *isomorphic*, if there exists a morphism $g : B \rightarrow A$ such that $g \circ f = id_A$ and $f \circ g = id_B$.

Definition B.4. A category $\mathbb{C}$ is *skeletal* if its only isomorphisms are the identity morphisms.

It is often useful to use the principle of *duality* to define two concepts at the same time using the same definition: one applies to category $\mathbb{C}$, the other applies to the dual category $\mathbb{C}^{op}$.

Definition B.5. For a category $\mathbb{C}$, its *dual category* $\mathbb{C}^{op}$ is a category with $Obj(\mathbb{C}^{op}) = Obj(\mathbb{C})$ and $Hom_{\mathbb{C}^{op}}(A, B) = Hom_{\mathbb{C}}(B, A)$, and the opposite composition $\hat{\circ}$ is defined for all $f : A \rightarrow B, g : B \rightarrow C$ in $\mathbb{C}^{op}$ as $g \hat{\circ} f = f \circ g$.

Definition B.6. For a category $\mathbb{C}$, its *discretisation* $|\mathbb{C}|$ is a category with all the objects of $\mathbb{C}$ and only identity morphisms.

There are several simple examples of categories:

- **Set:** a category whose objects are sets,¹ morphisms $f : A \rightarrow B$ are functions from A to B , and composition is function composition;
- **1:** the category with only one object and one morphism (identity);
- **2:** category with two objects and only two morphisms (identities associated with these objects).

B.1.2 Functors and natural transformations

Functors can be seen as a way of structurally mapping objects and morphisms from one category to another.

¹we can further specify which notion of ‘set’ we are using – here we stick to the usual notion of Zermelo-Fraenkel sets, with the axiom of choice

Definition B.7. For two categories $\mathbb{C}$ and $\mathbb{D}$, a *functor* $F : \mathbb{C} \rightarrow \mathbb{D}$ consists of a mapping of objects A in $\mathbb{C}$ to objects FA in $\mathbb{D}$ and a mapping of morphisms $f : A \rightarrow B$ to $Ff : FA \rightarrow FB$ such that identity morphisms map to identities ($Fid_A = id_{FA}$) and composition is respected ($F(g \circ f) = Fg \circ Ff$).

Functors can be composed: for $F : \mathbb{C} \rightarrow \mathbb{D}$, $G : \mathbb{D} \rightarrow \mathbb{E}$, the functor $G \bullet F$ (also written GF) exists and maps $\mathbb{C}$ to $\mathbb{E}$.

Natural transformations are a way of representing maps between functors with the same source and target categories.

Definition B.8. For two functors $F, G : \mathbb{C} \rightarrow \mathbb{D}$, a natural transformation $\alpha : F \Rightarrow G$ is a family of morphisms $\alpha_A : FA \rightarrow GA$ for all objects $A \in \mathbb{C}$ such that for all $A, B \in \mathbb{C}$, the following diagram commutes:

$$\begin{array}{ccc} FA & \xrightarrow{\alpha_A} & GA \\ Ff \downarrow & & \downarrow Gf \\ FB & \xrightarrow{\alpha_B} & GB \end{array}$$

Natural transformations can be composed, but there are two applicable notions. ‘Vertical composition’ addresses natural transformations $\alpha : F \Rightarrow G$, $\theta : G \Rightarrow H$ for $F, G, H : \mathbb{C} \rightarrow \mathbb{D}$, with composition $\beta \circ \alpha : F \Rightarrow H$ defined with $(\beta \circ \alpha)_A = \beta_A \circ \alpha_A$. By contrast, ‘horizontal composition’ deals with natural transformations $\alpha : F \Rightarrow G$ for $F, G : \mathbb{C} \rightarrow \mathbb{D}$ and $\beta : U \Rightarrow V$ for $U, V : \mathbb{D} \rightarrow \mathbb{E}$, and the composition is a natural transformation $\beta \star \alpha : UF \Rightarrow VG$ defined with $(\beta \star \alpha)_A = \beta_{GA} \circ U\alpha_A$. Both of these notions are associative and have an identity.

We sometimes define natural transformations and functors in multiple parameters. For example, a *bifunctor* $F : \mathbb{C} \times \mathbb{D} \rightarrow \mathbb{E}$ maps a pair of objects $A \in \mathbb{C}$ and $B \in \mathbb{D}$ to objects $FAB \in \mathbb{E}$, as well as pairs of morphisms $f : A \rightarrow A'$ in $\mathbb{C}$ and $g : B \rightarrow B'$ in $\mathbb{D}$ to morphisms Ffg , and furthermore ‘fixing’ one of the arguments gives a functor: for an object A and morphism $f : A \rightarrow A'$ in $\mathbb{C}$, there is a functor $G : \mathbb{D} \rightarrow \mathbb{E}$ defined as $GB = FAB$, $Gg = Ffg$, and likewise for the second argument.

B.1.3 Bicategories

What was presented here were categories: there are objects and morphisms between objects. We can further consider *bicategories*: aside from objects – also called *0-cells* – and morphisms – also called *1-cells* – there are now *2-cells* (2-morphisms), which can be seen as arrows from morphisms to morphisms. More formally, a bicategory $\mathbb{C}$ consists of a collection of 0-cells, for each pair of 0-cells A and B a category $\mathbb{C}(A, B)$ whose objects are called 1-cells and whose morphisms are called 2-cells, an identity 1-cell id_A for each 0-cell A , a *horizontal composition* functor $\circ : \mathbb{C}(B, C) \times \mathbb{C}(A, B) \rightarrow \mathbb{C}(A, C)$ for all 0-cells A, B, C , as well as natural isomorphisms called unitors and associators which we do not elaborate further. Bicategories are occasionally referenced in this thesis, but are not of major importance.

In diagrams, k -cells are visualised as arrows with k shafts and a single arrowhead.

A simple example of a 2-category is **Cat**, the 2-category of small categories. Its 0-cells are small categories, 1-cells from $\mathbb{C}$ to $\mathbb{D}$ are functors $F : \mathbb{C} \rightarrow \mathbb{D}$, and 2-cells from $F : \mathbb{C} \rightarrow \mathbb{D}$ to $G : \mathbb{C} \rightarrow \mathbb{D}$ are natural transformations $\alpha : F \Rightarrow G$. Composition

for 1-cels is just functor composition $\bullet$, whereas composition for 2-cels is ‘vertical’ composition of natural transformations.

$$\begin{array}{ccc}
 & F & \\
 \mathbb{C} & \begin{array}{c} \searrow \alpha \\ \downarrow \\ \searrow \beta \end{array} & \mathbb{D} \\
 & G & \\
 & H &
 \end{array}$$

B.2 Universal constructions

B.2.1 Initial and terminal object

In **Set**, we note that for any set A and any singleton set $\{*\}$, there is a unique function $A \rightarrow \{*\}$: one which maps all elements of A to $*$. This notion generalises to the concept of a *terminal object*.

Definition B.9. A *terminal object* in a category $\mathbb{C}$ is an object $1 \in \mathbb{C}$ such that for every object $A \in \mathbb{C}$ there is a unique morphism $!_A : A \rightarrow 1$.

It can be shown that all terminal objects in a category are isomorphic.

The dual notion of an *initial object* can be concisely defined as the terminal object in a dual category. Expanding this idea gives us the following definition:

Definition B.10. An *initial object* in a category $\mathbb{C}$ is an object $0 \in \mathbb{C}$ such that for every object $A \in \mathbb{C}$ there is a unique morphism $?_A : 0 \rightarrow A$.

In **Set**, the initial object is the empty set $\emptyset$.

B.2.2 Products and coproducts

The notion of *product* in category theory generalises the cartesian product in set theory, direct product of groups in group theory, as well as product topological spaces.

Definition B.11. For two objects A and B in a category $\mathbb{C}$, their *product* is an object $A \times B$, along with two *projection* morphisms $\pi_1^{A,B} : A \times B \rightarrow A$ and $\pi_2^{A,B} : A \times B \rightarrow B$ such that, for any other object $C \in \mathbb{C}$ with morphisms $f : C \rightarrow A$ and $g : C \rightarrow B$, there exists a unique morphism $\langle f, g \rangle$ such that $\pi_1^{A,B} \circ \langle f, g \rangle = f$ and $\pi_2^{A,B} \circ \langle f, g \rangle = g$ – in other words, the following diagram commutes:

$$\begin{array}{ccccc}
 & & C & & \\
 & f \swarrow & & \searrow g & \\
 A & & \langle f, g \rangle & & B \\
 & \nwarrow \pi_1^{A,B} & \downarrow & \nearrow \pi_2^{A,B} & \\
 & & A \times B & &
 \end{array}$$

We can then define the notion of *coproduct* concisely by saying that a coproduct of two objects A and B in $\mathbb{C}$ is the same as the product of A and B in $\mathbb{C}^{\text{op}}$. Expanding this definition, we get the following:

Definition B.12. For two objects A and B in a category $\mathbb{C}$, their *coproduct* is an object $A + B$, along with two *injection* morphisms $\iota_1^{A,B} : A \rightarrow A + B$ and $\iota_2^{A,B} : B \rightarrow A + B$ such that, for any other object $C \in \mathbb{C}$ with morphisms $f : A \rightarrow C$ and $g : B \rightarrow C$, there exists a unique morphism $[f, g]$ such that $[f, g] \circ \iota_1^{A,B} = f$ and $[f, g] \circ \iota_2^{A,B} = g$ – in other words, the following diagram commutes:

$$\begin{array}{ccc}
 & A + B & \\
 \iota_1^{A,B} \swarrow & \vdots & \searrow \iota_2^{A,B} \\
 A & & B \\
 f \searrow & \downarrow [f,g] & \swarrow g \\
 & C &
 \end{array}$$

Coproducts (sometimes also called *sums*) correspond to disjoint unions in set theory – more familiar to a reader with a programming background as sum types or enumerations.

We tend to consider categories with all finite product or with all finite coproducts. If a category has both all finite products and all finite coproducts, then there is a unique ‘canonical’ morphism $A \times B + A \times C \rightarrow A \times (B + C)$ for all objects A, B and C (following the definition of product and coproduct) If this morphism is an isomorphism, then the category is said to be *distributive*.

B.2.3 Exponentials

The notion of exponential in a category generalises function spaces for sets (or homomorphism spaces for groups and similar structures):

Definition B.13. Let $\mathbb{C}$ be a category with finite products and let A, B be objects of $\mathbb{C}$. The object B^A is an *exponential* if there exists a unique morphism $\text{ev}_{A,B} : B^A \times A \rightarrow B$ such that, for any object $C \in \mathbb{C}$ and morphism $f : C \times A \rightarrow B$, there is a unique morphism $\lambda f : C \rightarrow B^A$ such that the following diagram commutes:

$$\begin{array}{ccc}
 C \times A & & \\
 \langle \lambda f, \text{id}_A \rangle \downarrow & \searrow f & \\
 B^A \times A & \xrightarrow{\text{ev}_{A,B}} & B
 \end{array}$$

The morphism λf is sometimes also denoted $\text{curry}(f)$. We sometimes refer to the morphism $\text{ev}_{A,B}$ as *evaluation map*.

Definition B.14. A category $\mathbb{C}$ is *cartesian closed* (‘ccc’ is used as a shorthand) if it has a terminal object 1 , all finite products, and all exponentials.

Cartesian closed categories are the model used to define the semantics of the simply-typed lambda calculus.

B.2.4 Limits and colimits

Notions such as products and terminal objects can be seen as examples of a further, more general notion – that being the *limit*. We now define limits and their dual, *colimits*.

Definition B.15. Let $\mathbb{I}$ and $\mathbb{C}$ be categories and let $F : \mathbb{I} \rightarrow \mathbb{C}$ be a functor. A *cone* (X, χ) of F is an object $X \in \mathbb{C}$ together with a family $\chi_A : X \rightarrow FA$ of morphisms indexed by objects $A \in \mathbb{I}$, such that for every morphism $f : A \rightarrow B$ in $\mathbb{I}$, the equality $Ff \circ \chi_A = \chi_B$ holds. In other words, the following diagram commutes in $\mathbb{C}$ (for all $f : A \rightarrow B$ in $\mathbb{I}$):

$$\begin{array}{ccc} & X & \\ \chi_A \swarrow & & \searrow \chi_B \\ FA & \xrightarrow{Ff} & FB \end{array}$$

Dually, we can define *co-cones* as a dual of cones by inverting all morphisms in the above definition. The functor $F : \mathbb{I} \rightarrow \mathbb{C}$ is referred to as a *diagram of shape $\mathbb{I}$* .

A limit is a universal cone – the ‘terminal’ cone, in a sense. This is more elaborated with this following definition:

Definition B.16. Let $\mathbb{I}$ and $\mathbb{C}$ be categories and let $F : \mathbb{I} \rightarrow \mathbb{C}$ be a functor. A cone (L, κ) of F is a *limit* if, for any other cone (X, χ) , there exists a unique morphism $u : X \rightarrow L$ such that, for any $A \in \mathbb{I}$, $\chi_A = \kappa_A \circ u$. Combining with the definition of a cone, this means that the following diagram commutes in $\mathbb{C}$ for all $f : A \rightarrow B$ in $\mathbb{I}$:

$$\begin{array}{ccc} & X & \\ & \downarrow u & \\ & L & \\ \chi_A \swarrow & & \searrow \chi_B \\ FA & \xrightarrow{Ff} & FB \\ \kappa_A \swarrow & & \searrow \kappa_B \end{array}$$

A colimit is, dually, the initial co-cone, or a limit in $\mathbb{C}^{\text{op}}$.

A product can be seen as a limit in which the shape of the diagram is the two-object discrete category $\mathbf{2}$.

B.3 Monoidal categories

Significant parts of the underlying theory of (graded) monads rely on an understanding of the fundamentals of monoidal categories.

B.3.1 Laws and classification of monoidal categories

Definition B.17. A monoidal category $\mathcal{C}$ is a tuple $(\mathbb{C}, \otimes, I, \alpha, \lambda, \rho)$, where:

- $\mathbb{C}$ is a category;

- $\otimes$ is a bifunctor $\otimes : \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$ called a *tensor product*;
- I is an object $I \in \mathbb{C}$ called the *unit*;
- α is the *associator*, a natural isomorphism with three parameters, mapping $\alpha_{A,B,C} : A \otimes (B \otimes C) \rightarrow (A \otimes B) \otimes C$;
- λ is the *left unitor*, a natural isomorphism with components $\lambda_A : I \otimes A \rightarrow A$;
- ρ is the *right unitor*, a natural isomorphism with components $\rho_A : A \otimes I \rightarrow A$

such that the following diagrams commute:

$$\begin{array}{ccccc}
 A \otimes (B \otimes (C \otimes D)) & \xrightarrow{\alpha_{A,B,C \otimes D}} & (A \otimes B) \otimes (C \otimes D) & & \\
 \downarrow \text{id}_A \otimes \alpha_{B,C,D} & & \searrow \alpha_{A \otimes B, C, D} & & \\
 A \otimes ((B \otimes C) \otimes D) & \xrightarrow{\alpha_{A, B \otimes C, D}} & (A \otimes (B \otimes C)) \otimes D & \xrightarrow{\alpha_{A, B, C} \otimes \text{id}_D} & ((A \otimes B) \otimes C) \otimes D
 \end{array}$$

$$\begin{array}{ccc}
 A \otimes (I \otimes B) & \xrightarrow{\alpha_{A, I, B}} & (A \otimes I) \otimes B \\
 \searrow \text{id}_A \otimes \lambda_A & & \swarrow \rho_A \otimes \text{id}_B \\
 & A \otimes B &
 \end{array}$$

We tend to focus on monoidal categories in which associator and unitor natural transformations are all identities. We call such categories *strict*.

A simple example of a monoidal category is any category with finite products and terminal objects. Furthermore, every monoid in the traditional algebraic sense can be seen as a monoidal category, meaning a tuple $(E, \odot, \mathbf{1})$ corresponds to a discrete strict monoidal category where each object corresponds to an element of E , $\odot$ is the tensor product and $\mathbf{1}$ is the unit.

B.3.2 Monoids

We mention the definition of monoids in the context of monoidal categories because of their relationship to monads.

Definition B.18. A *monoid* in a monoidal category $(\mathbb{C}, \otimes, I, \alpha, \lambda, \rho)$ is a triple (M, μ, η) , where M is an object, with morphisms $\mu : M \otimes M \rightarrow M$ and $\eta : I \rightarrow M$:

$$\begin{array}{ccccc}
 M \otimes (M \otimes M) & \xrightarrow{\text{id}_M \otimes \mu} & M \otimes M & & \\
 \downarrow \alpha_{M, M, M} & & \downarrow \mu & & \\
 (M \otimes M) \otimes M & \xrightarrow{\mu \otimes \text{id}_M} & M \otimes M & \xrightarrow{\mu} & M
 \end{array}$$

$$\begin{array}{ccccc}
 I \otimes M & \xrightarrow{\eta \otimes \text{id}_M} & M \otimes M & \xleftarrow{\text{id}_M \otimes \eta} & M \otimes I \\
 & \searrow \lambda_M & \downarrow \mu & \swarrow \rho_M & \\
 & & M & &
 \end{array}$$

In **Set**, a monoid (M, μ, η) corresponds to an actual monoid, with M the carrier set, μ the binary multiplication operator, and η is a function $\{*\} \rightarrow M$ which ‘selects’ the unit (the unit is $\eta(*)$).

B.3.3 Monoidal functors and natural transformations

We can define functors and natural transformations on monoidal categories which ‘respect’ the monoidal structure. If a monoidal category generalises the idea of monoid, then functors between them generalise monoid homomorphisms:

Definition B.19. Given monoidal categories $\mathcal{C} = (\mathbb{C}, \otimes_{\mathbb{C}}, I_{\mathbb{C}}, \alpha^{\mathbb{C}}, \lambda^{\mathbb{C}}, \rho^{\mathbb{C}})$ and $\mathcal{D} = (\mathbb{D}, \otimes_{\mathbb{D}}, I_{\mathbb{D}}, \alpha^{\mathbb{D}}, \lambda^{\mathbb{D}}, \rho^{\mathbb{D}})$, a *lax monoidal functor* is a tuple (F, μ, η) , where $F : \mathbb{C} \rightarrow \mathbb{D}$ is a functor, μ is a natural transformation in two arguments, with components $\mu_{A,B} : FA \otimes_{\mathbb{D}} FB \rightarrow F(A \otimes_{\mathbb{C}} B)$, and η is a morphism $I_{\mathbb{D}} \Rightarrow FI_{\mathbb{C}}$ such that the following diagrams all commute:

$$\begin{array}{ccc}
 FA \otimes_{\mathbb{D}} (FB \otimes_{\mathbb{D}} FC) & \xrightarrow{\text{id}_{FA} \otimes \mu_{B,C}} & FA \otimes_{\mathbb{D}} F(B \otimes_{\mathbb{C}} C) \xrightarrow{\mu_{FA, F(B \otimes_{\mathbb{C}} C)}} F(A \otimes_{\mathbb{C}} (B \otimes_{\mathbb{C}} C)) \\
 \alpha_{FA, FB, FC}^{\mathbb{D}} \downarrow & & \downarrow F\alpha_{A,B,C}^{\mathbb{C}} \\
 (FA \otimes_{\mathbb{D}} FB) \otimes_{\mathbb{D}} FC & \xrightarrow{\mu_{A,B} \otimes \text{id}_{FC}} & F(A \otimes_{\mathbb{C}} B) \otimes_{\mathbb{D}} FC \xrightarrow{\mu_{F(A \otimes_{\mathbb{C}} B), FC}} F((A \otimes_{\mathbb{C}} B) \otimes_{\mathbb{C}} C)
 \end{array}$$

$$\begin{array}{ccc}
 I_{\mathbb{D}} \otimes_{\mathbb{D}} FA & \xrightarrow{\eta \otimes \text{id}_{FA}} & FI_{\mathbb{C}} \otimes_{\mathbb{D}} FA \\
 \lambda_{FA}^{\mathbb{D}} \downarrow & & \downarrow \mu_{I_{\mathbb{C}}, A} \\
 FA & \xleftarrow{F\lambda_A^{\mathbb{C}}} & F(I_{\mathbb{C}} \otimes_{\mathbb{C}} A)
 \end{array}
 \qquad
 \begin{array}{ccc}
 FA \otimes_{\mathbb{D}} I_{\mathbb{D}} & \xrightarrow{\text{id}_{FA} \otimes \eta} & FA \otimes_{\mathbb{D}} FI_{\mathbb{C}} \\
 \rho_{FA}^{\mathbb{D}} \downarrow & & \downarrow \mu_{A, I_{\mathbb{C}}} \\
 FA & \xleftarrow{F\rho_A^{\mathbb{C}}} & F(A \otimes_{\mathbb{C}} I_{\mathbb{C}})
 \end{array}$$

If μ is an identity transformation and η is an identity morphism, then the lax monoidal functor (lmf) is *strict*.

Definition B.20. Given two monoidal categories $\mathcal{C} = (\mathbb{C}, \otimes_{\mathbb{C}}, I_{\mathbb{C}}, \alpha^{\mathbb{C}}, \lambda^{\mathbb{C}}, \rho^{\mathbb{C}})$ and $\mathcal{D} = (\mathbb{D}, \otimes_{\mathbb{D}}, I_{\mathbb{D}}, \alpha^{\mathbb{D}}, \lambda^{\mathbb{D}}, \rho^{\mathbb{D}})$, and two lax monoidal functors $(F_1, \mu^1, \eta^1) : \mathcal{C} \rightarrow \mathcal{D}$ and $(F_2, \mu^2, \eta^2) : \mathcal{C} \rightarrow \mathcal{D}$, a *monoidal natural transformation* is a natural transformation $f : F_1 \Rightarrow F_2$ such that the following diagrams commute for all $A, B \in \mathbb{C}$:

$$\begin{array}{ccc}
 F_1 A \otimes_{\mathbb{D}} F_1 B & \xrightarrow{f_A \otimes f_B} & F_2 A \otimes_{\mathbb{D}} F_2 B \\
 \mu_{A,B}^1 \downarrow & & \downarrow \mu_{A,B}^2 \\
 F_1(A \otimes_{\mathbb{C}} B) & \xrightarrow{f_{A \otimes_{\mathbb{C}} B}} & F_2(A \otimes_{\mathbb{C}} B)
 \end{array}$$

$$\begin{array}{ccc} & I_{\mathbb{D}} & \\ \eta^1 \swarrow & & \searrow \eta^2 \\ F_1 I_{\mathbb{C}} & \xrightarrow{f_{\mathbb{C}}} & F_2 I_{\mathbb{C}} \end{array}$$

Haskell code listings

In this appendix, I provide the full implementationss of Haskell embeddings of some of the graded monads mentioned in the main part of the thesis. This includes the `analysis-monad` library, which acts as a practical contribution of this thesis. While for the most part it contains code listings of the implementations of the analysis-graded monads of the libraries, there are a few brief tutorial-like notes on usage and justifications for some of the design choices.

C.1 Analysis-graded monads

C.1.1 Usage example

A simple way of using the analysis-graded version of `Maybe` can be seen in the following example:

```
-- foo has type GMaybe T Int
foo b = if b then return 1 else return 2
-- bar has type GMaybe U Int
bar b = if b then return 1 else GNothing
-- baz has type GMaybe F Int
baz b = bar b >> GNothing
```

The syntax for programming with analysis-graded monads matches the syntax familiar to programmers using monads, in particular the binds and returns.

C.1.2 Components of analysis-graded monads

Source code

```
{-# LANGUAGE AllowAmbiguousTypes    #-}
{-# LANGUAGE ConstraintKinds         #-}
{-# LANGUAGE DataKinds               #-}
{-# LANGUAGE FlexibleContexts        #-}
{-# LANGUAGE MultiParamTypeClasses   #-}
{-# LANGUAGE PolyKinds                #-}
{-# LANGUAGE TypeFamilies             #-}
{-# LANGUAGE TypeOperators           #-}
{-# LANGUAGE UndecidableInstances     #-}
```

```
module Control.Granted where
```

```
import GHC.Exts (Constraint)
import qualified Prelude as P
import Prelude hiding (Monad(..))
```

```

class GFunctor (f :: k -> * -> *) where
  fmapG :: (a -> b) -> f r a -> f r b

class GMonad m => GBindable (m :: k -> * -> *)
  (r :: k) (s :: k) where
  bindG :: m r a -> (a -> m s b) -> m (Seq m r s) b

type LeftId m e = ((Seq m e (Unit m)) ~ e)
type RightId m e = ((Seq m (Unit m) e) ~ e)
type Assoc m e f g = ((Seq m (Seq m e f) g)
  ~ (Seq m e (Seq m f g)))
type JoinIdemp m e = ((Join m e e) ~ e)
type JoinComm m e f = ((Join m e f) ~ (Join m f e))
type JoinUpper m e f = (Subeffect m e (Join m e f),
  Subeffect m f (Join m e f))
type Refl m e = Subeffect m e e
type Trans m e f g = (Subeffect m e f, Subeffect m f g,
  Subeffect m e g)
type Monotonic m e e' f f' = (Subeffect m e e', Subeffect m f f',
  Subeffect m (Seq m e f)
    (Seq m e' f'))

class GFunctor m => GMonad (m :: k -> * -> *) where
  type Unit m :: k
  type Seq m (e :: k) (f :: k) :: k
  type Join m (e :: k) (f :: k) :: k

  type Inv m (f :: k) (g :: k) :: Constraint
  type Inv m f g = ()

  return :: a -> m (Unit m) a

  (>=>) :: GBindable m e f => m e a -> (a -> m f b)
    -> m (Seq m e f) b
  x >=> k = bindG x k
  (>>) :: GBindable m e f => m e a -> m f b
    -> m (Seq m e f) b
  x >> y = x >=> const y

  alt :: JoinUpper m e f => Bool -> m e a -> m f a
    -> m (Join m e f) a
  alt True x y = sub x
  alt False x y = sub y

class Subeffect (m :: k -> * -> *) (e :: k) (f :: k) where
  sub :: m e a -> m f a

```

Explanation

Properties used to guarantee that the Haskell type system is capable of evaluating type families and making desired inferences are encoded as type equality constraints. For example: `LeftId m e` expresses the fact that sequencing the grade `e` with the unit grade corresponding to the analysis-graded monad `m` results in `e`; `Refl` expresses the reflexivity property of the subeffecting relation.

A `GMonad` instance needs to be a `GFunctor` instance, similarly to what is required of `Monad` instances (they also need to be instances of `Functor`).

The `bind` operator requires the use of the `GBindable` type class in order to allow the full definition to be written using pattern matching (ad-hoc polymorphism needs to be exploited as it is impossible to pattern-match on a function-typed argument).

The `alt` function is a simple case of a ‘graded conditional’ construct: the resulting computation is one of the two branches, and the resulting grade is the join of the grades of the two computations.

C.1.3 Stateful computation

In the `analysis-monad` library, there are several analysis-graded monads used for representing stateful computation. Following the conventions established with stateful monads, manipulating states using `getG/putG` computations is enabled using a `GradedState` type class (serving the same purpose as `MonadState` in core Haskell). A stateful analysis-graded monad needs to contain grades representing reads, grades representing writes, `get` and `put` operators.

Permissions

```
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE TypeFamilies        #-}
```

```
module Control.Graded.PermState where
```

```
import Control.Graded
```

```
data Perm = Pure | RO | AW | WO | RW
```

```
type family PermJoin r s where
```

```
  PermJoin s s = s
  PermJoin Pure AW = WO
  PermJoin Pure s = s
  PermJoin AW Pure = WO
  PermJoin s Pure = s
  PermJoin AW WO = WO
  PermJoin WO AW = WO
  PermJoin _ _ = RW
```

```
type family PermSeq r s where
```

```
  PermSeq s s = s
  PermSeq Pure s = s
  PermSeq s Pure = s
  PermSeq AW _ = AW
  PermSeq RO _ = RW
  PermSeq RW _ = RW
  PermSeq WO RO = RW
  PermSeq WO RW = RW
  PermSeq WO AW = AW
```

```

data PermState s (f :: Perm) a where
  PureState :: {runPureState :: a}      -> PermState s Pure a
  ROState   :: {runROState   :: s -> a} -> PermState s RO a
  AWState   :: {runAWState   :: (a, s)} -> PermState s AW a
  WOSTate   :: {runWOSTate   :: (a, Maybe s)} -> PermState s WO a
  RWState   :: {runRWState   :: s -> (a, s)} -> PermState s RW a

runPermState :: PermState s f a -> s -> (a, s)
runPermState (PureState x)    s = (x, s)
runPermState (ROState f)      s = (f s, s)
runPermState (AWState (x, s)) _ = (x, s)
runPermState (WOSTate (x, m)) s = case m of
  Nothing -> (x, s)
  Just y  -> (x, y)
runPermState (RWState f)      s = f s

instance Subeffect (PermState s) r r where
  sub = id
instance Subeffect (PermState s) Pure RO where
  sub (PureState x) = ROState (const x)
instance Subeffect (PermState s) Pure WO where
  sub (PureState x) = WOSTate (x, Nothing)
instance Subeffect (PermState s) Pure RW where
  sub (PureState x) = RWState (\s -> (x, s))
instance Subeffect (PermState s) AW WO where
  sub (AWState (x, s)) = WOSTate (x, Just s)
instance Subeffect (PermState s) AW RW where
  sub (AWState (x, s)) = RWState (const (x, s))
instance Subeffect (PermState s) RO RW where
  sub (ROState x) = RWState (\s -> (x s, s))
instance Subeffect (PermState s) WO RW where
  sub (WOSTate (x, Just s)) = RWState (const (x, s))
  sub (WOSTate (x, Nothing)) = RWState (\s -> (x, s))

instance GBindable (PermState s) Pure r where
  bindG (PureState x) k = k x
instance GBindable (PermState s) RO Pure where
  bindG (ROState f) k = ROState (runPureState . k . f)
instance GBindable (PermState s) RO RO where
  bindG (ROState f) k = ROState (\s -> (runROState (k (f s))) s)
instance GBindable (PermState s) RO AW where
  bindG (ROState f) k = RWState (runAWState . k . f)
instance GBindable (PermState s) RO WO where
  bindG (ROState f) k = RWState (\s ->
    let (r, s') = runWOSTate (k (f s)) in case s' of
      Just y -> (r, y)
      Nothing -> (r, s))
instance GBindable (PermState s) RO RW where
  bindG (ROState f) k = RWState (\s -> runRWState (k (f s)) s)
instance GBindable (PermState s) AW Pure where
  bindG (AWState (x, s)) k = AWState (runPureState (k x), s)
instance GBindable (PermState s) AW RO where
  bindG (AWState (x, s)) k = AWState (runROState (k x) s, s)
instance GBindable (PermState s) AW AW where
  bindG (AWState (x, s)) k = k x
instance GBindable (PermState s) AW WO where

```

```

    bindG (AWState (x, s)) k = AWState
      (let (r, s') = runWOSTate (k x) in case s' of
        Just y -> (r, y)
        Nothing -> (r, s))
instance GBindable (PermState s) AW RW where
  bindG (AWState (x, s)) k = AWState (runRWState (k x) s)
instance GBindable (PermState s) WO Pure where
  bindG (WOSTate (x, s)) k = WOSTate (runPureState (k x), s)
instance GBindable (PermState s) WO RO where
  bindG (WOSTate (x, Just s)) k = RWState (\s' ->
    (runROState (k x) s, s))
  bindG (WOSTate (x, Nothing)) k = RWState (\s' ->
    (runROState (k x) s', s'))
instance GBindable (PermState s) WO AW where
  bindG (WOSTate (x, _)) k = k x
instance GBindable (PermState s) WO WO where
  bindG (WOSTate (x, Just s)) k = WOSTate
    (let (r, s') = runWOSTate (k x) in case s' of
      Just y -> (r, Just y)
      Nothing -> (r, Just s))
  bindG (WOSTate (x, Nothing)) k = k x
instance GBindable (PermState s) WO RW where
  bindG (WOSTate (x, Just s)) k = RWState (\s' ->
    runRWState (k x) s)
  bindG (WOSTate (x, Nothing)) k = k x
instance GBindable (PermState s) RW Pure where
  bindG (RWState f) k = RWState (\s ->
    let (x, s') = f s in (runPureState (k x), s'))
instance GBindable (PermState s) RW RO where
  bindG (RWState f) k = RWState (\s ->
    let (x, s') = f s in (runROState (k x) s', s'))
instance GBindable (PermState s) RW AW where
  bindG (RWState f) k = RWState (\s ->
    let (x, _) = f s in runAWState (k x))
instance GBindable (PermState s) RW WO where
  bindG (RWState f) k = RWState (\s ->
    let (x, s') = f s in let (r, s'') = runWOSTate (k x)
      in case s'' of
        Just y -> (r, y)
        Nothing -> (r, s'))
instance GBindable (PermState s) RW RW where
  bindG (RWState f) k = RWState (\s ->
    let (x, s') = f s in runRWState (k x) s')

instance GFunctor (PermState s) where
  fmapG f (PureState x) = PureState (f x)
  fmapG f (ROState f') = ROState (\s -> f (f' s))
  fmapG f (AWState (x, s)) = AWState (f x, s)
  fmapG f (WOSTate (x, s)) = WOSTate (f x, s)
  fmapG f (RWState f') = RWState (\s ->
    (\(x, y) -> (f x, y)) (f' s))

instance GMonad (PermState s) where
  type Unit (PermState s) = Pure
  type Seq (PermState s) e f = PermSeq e f
  type Join (PermState s) e f = PermJoin e f

```

```

    return = PureState

get :: PermState s R0 s
get = R0State id

put :: s -> PermState s AW ()
put s = AWState ((), s)

```

History

```

{-# LANGUAGE DataKinds #-}
{-# LANGUAGE GADTs #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds #-}
{-# LANGUAGE TypeOperators #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE UndecidableInstances #-}

module Control.Graded.HistState where

import Prelude as P hiding (Monad(..))
import Control.Graded

data Nat = Succ Nat | Zero

class NatLE (m :: Nat) (n :: Nat)
instance NatLE Zero (Succ n)
instance NatLE m n => NatLE (Succ m) (Succ n)

data a :~: b where
    Refl :: a :~: a

data NatI (n :: Nat) where
    ZI :: NatI Zero
    SI :: NatI n -> NatI (Succ n)

data Vec (n :: Nat) a where
    VNil :: Vec Zero a
    (:::) :: {vHead :: a, vTail :: Vec n a} -> Vec (Succ n) a

infixr 5 :::

data VecTo (n :: Nat) a where
    FromVec :: Vec n a -> VecTo n a
    Ext      :: {unExt :: VecTo n a} -> VecTo (Succ n) a

vecAsList :: Vec n a -> [a]
vecAsList VNil = []
vecAsList (x:::xs) = x : vecAsList xs

vecToList :: VecTo n a -> [a]
vecToList (FromVec v) = vecAsList v
vecToList (Ext vecto) = vecToList vecto

data NatInf = Fin Nat | Inf

```

```

data NatInfI (n :: NatInf) where
  FinI :: NatI m -> NatInfI (Fin m)
  InfI :: NatInfI Inf

type family AddNat (m :: Nat) (n :: Nat) :: Nat where
  AddNat Zero n      = n
  AddNat (Succ m) n = Succ (AddNat m n)

-- inductive proof of: n + Z = n
sumZeroRight :: NatI n -> (AddNat n Zero) :~: n
sumZeroRight ZI = Refl
sumZeroRight (SI n') = case sumZeroRight n' of
  Refl -> Refl

-- inductive proof of: n + S m = S (n + m)
sumSuccRight :: NatI n -> NatI m
               -> (AddNat n (Succ m)) :~: Succ (AddNat n m)
sumSuccRight ZI _ = Refl
sumSuccRight (SI n') m = case sumSuccRight n' m of
  Refl -> Refl

-- inductive proof of commutativity: n + m = m + n
sumComm :: NatI n -> NatI m
         -> (AddNat n m) :~: (AddNat m n)
sumComm ZI m = case sumZeroRight m of Refl -> Refl
sumComm (SI n') m = case (sumComm n' m, sumSuccRight m n') of
  (Refl, Refl) -> Refl

-- inductive proof of commutativity of inf: n + m = m + n
sumCommInf :: NatInfI n -> NatInfI m
            -> (AddInf n m) :~: (AddInf m n)
sumCommInf InfI _ = Refl
sumCommInf _ InfI = Refl
sumCommInf (FinI m) (FinI n) = case sumComm m n of
  Refl -> Refl

vecLengthI :: Vec n a -> NatI n
vecLengthI VNil = ZI
vecLengthI (_:::xs) = SI (vecLengthI xs)

vecToLengthI :: VecTo n a -> NatI n
vecToLengthI (FromVec l) = vecLengthI l
vecToLengthI (Ext l)     = SI (vecToLengthI l)

(+++) :: Vec m a -> Vec n a -> Vec (AddNat m n) a
VNil    +++ l = l
(x:::xs) +++ l = x:::(xs +++ l)

infixr 5 +++

(<+++>) :: VecTo m a -> VecTo n a -> VecTo (AddNat m n) a
(FromVec xs) <+++> (FromVec ys) = FromVec (xs +++ ys)
(FromVec xs) <+++> (Ext ys)     =
  case sumSuccRight (vecLengthI xs) (vecToLengthI ys) of
    Refl -> Ext ((FromVec xs) <+++> ys)

```

```

(Ext xs)      <+> vec      = Ext (xs <+> vec)

infixr 5 <+>

type family AddInf (m :: NatInf) (n :: NatInf) :: NatInf where
  AddInf n Inf      = Inf
  AddInf Inf n      = Inf
  AddInf (Fin m) (Fin n) = Fin (AddNat m n)

type family MaxNat (m :: Nat) (n :: Nat) :: Nat where
  MaxNat Zero n      = n
  MaxNat m Zero      = m
  MaxNat (Succ m) (Succ n) = Succ (MaxNat m n)

type family JoinInf (m :: NatInf) (n :: NatInf) :: NatInf where
  JoinInf Inf _      = Inf
  JoinInf _ Inf      = Inf
  JoinInf (Fin m) (Fin n) = Fin (MaxNat m n)

data HistState s (f :: NatInf) a where
  FinHistState :: {runFinHistState :: s -> (a, VecTo n s)}
    -> HistState s (Fin n) a
  InfHistState :: {runInfHistState :: s -> (a, [s])}
    -> HistState s Inf a

runHistState :: HistState s n a -> s -> (a, [s])
runHistState (FinHistState c) s = let (r, l) = c s in
    (r, vecToList l)
runHistState (InfHistState c) s = c s

class Paddable (m :: Nat) (n :: Nat) where
  padExt :: VecTo m a -> VecTo n a

instance Paddable n n where
  padExt v = v
instance Paddable (Succ m) n => Paddable m n where
  padExt (Ext v) = padExt v

instance Subeffect (HistState s) r r where
  sub = id
instance Subeffect (HistState s) (Fin n) Inf where
  sub (FinHistState c) = InfHistState $
    \s -> let (r, l) = c s in (r, vecToList l)
instance (NatLE m n, Paddable m n)
=> Subeffect (HistState s) (Fin m) (Fin n) where
  sub (FinHistState c) = FinHistState $
    \s -> let (r, l) = c s in (r, padExt l)

instance GBindable (HistState s) (Fin m) (Fin n) where
  bindG (FinHistState c) k = FinHistState (\s ->
    let (x, l1) = c s in
    let FinHistState c' = k x in
    let (r, l2) = c' (head (vecToList l1 ++ [s])) in
    (r, case sumCommInf (FinI (vecToLengthI l2))
      (FinI (vecToLengthI l1)) of
      Refl -> l2 <+> l1))

```

```

instance GBindable (HistState s) (Fin m) Inf where
  bindG (FinHistState c) k = InfHistState (\s ->
    let (x, l1) = c s in
    let InfHistState c' = k x in
    let (r, l2) = c' (head (vecToList l1 ++ [s])) in
    (r, l2 ++ vecToList l1))
instance GBindable (HistState s) Inf (Fin n) where
  bindG (InfHistState c) k = InfHistState (\s ->
    let (x, l1) = c s in
    let FinHistState c' = k x in
    let (r, l2) = c' (head (l1 ++ [s])) in
    (r, vecToList l2 ++ l1))
instance GBindable (HistState s) Inf Inf where
  bindG (InfHistState c) k = InfHistState (\s ->
    let (x, l1) = c s in
    let InfHistState c' = k x in
    let (r, l2) = c' (head (l1 ++ [s])) in
    (r, l2 ++ l1))

instance GFunctor (HistState s) where
  fmapG f (FinHistState c) = FinHistState $
    \s -> let (r, l) = c s in (f r, l)
  fmapG f (InfHistState c) = InfHistState $
    \s -> let (r, l) = c s in (f r, l)

instance GMonad (HistState s) where
  type Unit (HistState s) = Fin Zero
  type Seq (HistState s) e f = AddInf f e
  type Join (HistState s) e f = JoinInf e f
  return x = FinHistState (\s -> (x, FromVec VNil))

nopHistState :: HistState s (Fin Zero) ()
nopHistState = return ()

```

GradedState

```

{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE UndecidableInstances #-}

module Control.Graded.GradedState where

import Prelude as P hiding (Monad(..))
import Control.Graded
import Control.Graded.Identity
import Control.Graded.Maybe
import Control.Graded.HistState
import Control.Graded.PermState
import Control.Graded.Trans.Identity
import Control.Graded.Trans.Maybe
import Control.Graded.Trans.PermState

```

```

class GMonad m => GradedState (s :: *) (m :: k -> * -> *)
    | m -> s where
    type Rd m :: k
    type Wr m :: k
    getG :: m (Rd m) s
    putG :: s -> m (Wr m) ()

instance GMonad m => GradedState s (PermStateT s m) where
    type Rd (PermStateT s m) = '(R0, Unit m)
    type Wr (PermStateT s m) = '(AW, Unit m)
    getG = R0StateT return
    putG s = AWStateT (return ((), s))

instance GradedState s (PermState s) where
    type Rd (PermState s) = R0
    type Wr (PermState s) = AW
    getG = R0State id
    putG s = AWState ((), s)

instance GradedState s (HistState s) where
    type Rd (HistState s) = Fin Zero
    type Wr (HistState s) = Fin (Succ Zero)
    getG = FinHistState (\s -> (s, FromVec VNil))
    putG s = FinHistState (\_ -> ((), FromVec (s:::VNil)))

instance GradedState s m => GradedState s (IdentityT m) where
    type Rd (IdentityT m) = '(S, Rd m)
    type Wr (IdentityT m) = '(S, Wr m)
    getG = IdentityT getG
    putG = IdentityT . putG

instance GradedState s m => GradedState s (GMaybeT m) where
    type Rd (GMaybeT m) = '(T, Rd m)
    type Wr (GMaybeT m) = '(T, Wr m)
    getG = GMaybeT (fmapG GJust getG)
    putG x = GMaybeT (fmapG GJust (putG x))

```

C.1.4 Other analysis-graded monads

Identity

```

{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeFamilies        #-}

module Control.Graded.Identity where

import Control.Graded
import Prelude hiding (Monad(..))
import qualified Prelude as P

data Sing = S

```

```

data Identity (r :: Sing) a where
  Identity :: {runIdentity :: a} -> Identity S a

instance Show a => Show (Identity S a) where
  show (Identity x) = show x

instance Subeffect Identity S S where
  sub = id

instance GFunctor Identity where
  fmapG f (Identity x) = Identity (f x)

instance GBindable Identity S S where
  bindG (Identity x) k = k x

instance GMonad (Identity :: Sing -> * -> *) where
  type Unit Identity = S
  type Seq Identity e f = S
  type Join Identity e f = S
  return = Identity

```

Monad

```

{-# LANGUAGE GADTs                #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeFamilies          #-}

module Control.Granted.Monad where

import Control.Granted
import Prelude hiding (Monad(..))
import qualified Prelude as P

data Monad m r a where
  Wrap :: P.Monad m => m a -> Monad m () a

  unwrap :: P.Monad m => Monad m r a -> m a
  unwrap (Wrap m) = m

instance (P.Monad m, Show (m a)) => Show (Monad m r a) where
  show (Wrap m) = show m

instance (P.Monad m) => Subeffect (Monad m) () () where
  sub = id

instance (P.Monad m) => GFunctor (Monad m) where
  fmapG f (Wrap m) = Wrap (fmap f m)

instance (P.Monad m) => GBindable (Monad m) () () where
  bindG (Wrap x) k = Wrap ((P.>=) x (unwrap . k))

instance P.Monad m => GMonad (Monad m) where
  type Unit (Monad m) = ()
  type Seq (Monad m) e f = ()
  type Join (Monad m) e f = ()
  return x = Wrap (P.return x)

```

Maybe

```

{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeFamilies        #-}

module Control.Granted.Maybe where

import Control.Granted

data MaybeGr = T | F | U

type family MaybeMult r s where
  MaybeMult F _ = F
  MaybeMult T s = s
  MaybeMult U F = F
  MaybeMult U _ = U

type family MaybeJoin r s where
  MaybeJoin s s = s
  MaybeJoin U _ = U
  MaybeJoin _ U = U
  MaybeJoin _ _ = U

data GMaybe r a where
  GNothing :: GMaybe F a
  GJust    :: a -> GMaybe T a
  GDyn     :: GMaybe r a -> GMaybe U a

runGMaybe :: GMaybe r a -> Maybe a
runGMaybe GNothing = Nothing
runGMaybe (GJust x) = Just x
runGMaybe (GDyn x) = runGMaybe x

instance Show a => Show (GMaybe r a) where
  show (GJust x) = "GJust_" ++ show x
  show GNothing = "GNothing"
  show (GDyn x) = show x

instance Subeffect GMaybe r r where
  sub = id
instance Subeffect GMaybe T U where
  sub = GDyn
instance Subeffect GMaybe F U where
  sub = GDyn

instance GFunctor GMaybe where
  fmapG f GNothing = GNothing
  fmapG f (GJust x) = GJust (f x)
  fmapG f (GDyn x) = GDyn (fmapG f x)

instance GBindable GMaybe F s where
  bindG GNothing _ = GNothing
instance GBindable GMaybe T s where

```

```

    bindG (GJust x) k = k x
instance GBindable GMaybe U F where
    bindG (GDyn x) _ = GNothing
instance GBindable GMaybe U T where
    bindG (GDyn GNothing) _ = GDyn GNothing
    bindG (GDyn (GJust x)) k = GDyn (k x)
instance GBindable GMaybe U U where
    bindG (GDyn GNothing) _ = GDyn GNothing
    bindG (GDyn (GJust x)) k = GDyn (k x)

instance GMonad GMaybe where
    type Unit GMaybe = T
    type Seq GMaybe e f = MaybeMult e f
    type Join GMaybe e f = MaybeJoin e f
    return x = GJust x

```

C.2 Graded monad transformers

Analysis-graded monads as defined in this library allow us to use facilities for programming using graded monad transformers. Graded monad transformers are all instances of the `GMonadTrans` type class and require defining the new grading algebra, as well as a graded lifting operator. They are described in more detail in Chapter 7.

C.2.1 Definition

```

{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE KindSignatures      #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE TypeFamilies        #-}

module Control.Graded.Trans where

import Control.Graded

class GMonadTrans (t :: (k -> * -> *) -> (k', k) -> * -> *) where
    type IdG t :: k'
    liftG      :: GMonad m => m s a -> t m '(IdG t, s) a

```

C.2.2 Instances of transformers

Identity

```

{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE FlexibleContexts     #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE TypeFamilies        #-}

module Control.Graded.Trans.Identity where

import Prelude hiding (Monad(..))

```

```

import Control.Graded
import Control.Graded.Identity
import Control.Graded.Trans

data IdentityT m (r :: (Sing, k)) a where
  IdentityT :: {runIdentityT :: m f a}
             -> IdentityT m '(S, f) a

instance (Show (m f a)) => Show (IdentityT m '(S, f) a) where
  show (IdentityT x) = show x

instance Subeffect m f f'
=> Subeffect (IdentityT m) '(S, f) '(S, f') where
  sub (IdentityT x) = IdentityT (sub x)

instance GFunctor m => GFunctor (IdentityT m) where
  fmapG f (IdentityT x) = IdentityT (fmapG f x)

instance GBindable m f f'
=> GBindable (IdentityT m) '(S, f) '(S, f') where
  bindG (IdentityT x) k = IdentityT (x >>= (runIdentityT . k))

instance GMonad m
=> GMonad (IdentityT m :: (Sing, k) -> * -> *) where
  type Unit (IdentityT m) = '(S, Unit m)
  type Seq (IdentityT m) '(S, f) '(S, f') = '(S, Seq m f f')
  type Join (IdentityT m) '(S, f) '(S, f') = '(S, Join m f f')
  return x = IdentityT (return x)

instance GMonadTrans IdentityT where
  type IdG IdentityT = S
  liftG = IdentityT

```

State with permissions

```

{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE FlexibleContexts     #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE UndecidableInstances #-}

module Control.Graded.Trans.PermState where

import Control.Graded
import Control.Graded.PermState
import Control.Graded.Trans
import Prelude hiding (Monad(..))

data PermStateT s (m :: k -> * -> *) (f :: (Perm, k)) a where
  PureStateT :: {runPureStateT :: m r a}
              -> PermStateT s m '(Pure, r) a
  ROStateT   :: {runROStateT   :: s -> m r a}
              -> PermStateT s m '(RO,   r) a

```

```

AWStateT    :: {runAWStateT    :: m r (a, s)}
              -> PermStateT s m '(AW, r) a
WStateT     :: {runWStateT     :: m r (a, Maybe s)}
              -> PermStateT s m '(WO, r) a
RWStateT    :: {runRWStateT    :: s -> m r (a, s)}
              -> PermStateT s m '(RW, r) a

runPermStateT :: GMonad m
              => PermStateT s m '(p, r) a -> s -> m r (a, s)
runPermStateT (PureStateT m) s = fmapG (\x -> (x, s)) m
runPermStateT (ROStateT f)    s = fmapG (\x -> (x, s)) (f s)
runPermStateT (AWStateT m)    _ = m
runPermStateT (WStateT m)     s = fmapG (\(x, s') -> case s' of
  Nothing -> (x, s)
  Just y   -> (x, y)) m
runPermStateT (RWStateT f)    s = f s

instance Subeffect m f f'
=> Subeffect (PermStateT s m) '(Pure, f) '(Pure, f') where
  sub (PureStateT m) = PureStateT (sub m)
instance Subeffect m f f'
=> Subeffect (PermStateT s m) '(AW, f) '(AW, f') where
  sub (AWStateT m) = AWStateT (sub m)
instance Subeffect m f f'
=> Subeffect (PermStateT s m) '(WO, f) '(WO, f') where
  sub (WStateT m) = WStateT (sub m)
instance Subeffect m f f'
=> Subeffect (PermStateT s m) '(RO, f) '(RO, f') where
  sub (ROStateT f) = ROStateT (\s -> sub (f s))
instance Subeffect m f f'
=> Subeffect (PermStateT s m) '(RW, f) '(RW, f') where
  sub (RWStateT f) = RWStateT (\s -> sub (f s))

instance Subeffect m f f'
=> Subeffect (PermStateT s m) '(Pure, f) '(RO, f') where
  sub (PureStateT m) = ROStateT $ const (sub m)
instance (Functor (m f), Subeffect m f f')
=> Subeffect (PermStateT s m) '(Pure, f) '(WO, f') where
  sub (PureStateT m) = WStateT $
    sub (fmap (\x -> (x, Nothing)) m)
instance (Functor (m f), Subeffect m f f')
=> Subeffect (PermStateT s m) '(Pure, f) '(RW, f') where
  sub (PureStateT m) = RWStateT $
    \s -> sub (fmap (\x -> (x, s)) m)
instance (Functor (m f), Subeffect m f f')
=> Subeffect (PermStateT s m) '(AW, f) '(WO, f') where
  sub (AWStateT m) = WStateT $
    sub (fmap (\(x, s) -> (x, Just s)) m)
instance (Functor (m f), Subeffect m f f')
=> Subeffect (PermStateT s m) '(AW, f) '(RW, f') where
  sub (AWStateT m) = RWStateT $
    const $ sub (fmap (\(x, s) -> (x, s)) m)
instance (Functor (m f), Subeffect m f f')
=> Subeffect (PermStateT s m) '(RO, f) '(RW, f') where
  sub (ROStateT f) = RWStateT $
    \s -> sub (fmap (\x -> (x, s)) (f s))

```

```

instance (Functor (m f), Subeffect m f f')
=> Subeffect (PermStateT s m) '(WO, f) '(RW, f') where
  sub (WOSTateT m) = RWStateT $
    \s' -> sub (fmap (\r -> case r of
      (x, Just s) -> (x, s)
      (x, Nothing) -> (x, s')) m)

instance GBindable m f f'
=> GBindable (PermStateT s m) '(Pure, f) '(Pure, f') where
  bindG (PureStateT x) k = PureStateT $
    x >>= (runPureStateT . k)
instance GBindable m f f'
=> GBindable (PermStateT s m) '(Pure, f) '(RO, f') where
  bindG (PureStateT x) k = ROStateT $
    \s -> x >>= \r -> runROStateT (k r) s
instance GBindable m f f'
=> GBindable (PermStateT s m) '(Pure, f) '(AW, f') where
  bindG (PureStateT x) k = AWStateT $
    x >>= (runAWStateT . k)
instance GBindable m f f'
=> GBindable (PermStateT s m) '(Pure, f) '(WO, f') where
  bindG (PureStateT x) k = WOSTateT $
    x >>= (runWOSTateT . k)
instance GBindable m f f'
=> GBindable (PermStateT s m) '(Pure, f) '(RW, f') where
  bindG (PureStateT x) k = RWStateT $
    \s -> (x >>= \r -> runRWStateT (k r) s)

instance GBindable m f f'
=> GBindable (PermStateT s m) '(RO, f) '(Pure, f') where
  bindG (ROStateT f) k = ROStateT $
    \s -> f s >>= (runPureStateT . k)
instance GBindable m f f'
=> GBindable (PermStateT s m) '(RO, f) '(RO, f') where
  bindG (ROStateT f) k = ROStateT $
    \s -> f s >>= \r -> runROStateT (k r) s
instance GBindable m f f'
=> GBindable (PermStateT s m) '(RO, f) '(AW, f') where
  bindG (ROStateT f) k = RWStateT $
    \s -> f s >>= (runAWStateT . k)
instance (GBindable m f f', GBindable m (Seq m f f') (Unit m),
  LeftId m (Seq m f f'))
=> GBindable (PermStateT s m) '(RO, f) '(WO, f') where
  bindG (ROStateT f) k = RWStateT $
    \s -> f s >>= (runWOSTateT . k) >>= \r ->
      let (y, s') = r in case s' of
        Just z -> return (y, z)
        Nothing -> return (y, s)
instance (GMonad m, GBindable m f f')
=> GBindable (PermStateT s m) '(RO, f) '(RW, f') where
  bindG (ROStateT f) k = RWStateT $
    \s -> f s >>= \r -> runRWStateT (k r) s

instance (GMonad m, GBindable m f f',
  GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(AW, f) '(Pure, f') where

```

```

    bindG (AWStateT m) k = AWStateT $
        m >>= \ (x, s) -> runPureStateT (k x) >>= \ r -> return (r, s)
instance (GMonad m, GBindable m f f',
          GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(AW, f) '(RO, f') where
    bindG (AWStateT m) k = AWStateT $
        m >>= \ (x, s) -> runROStateT (k x) s >>= \ r -> return (r, s)
instance (GMonad m, GBindable m f f',
          GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(AW, f) '(AW, f') where
    bindG (AWStateT m) k = AWStateT $
        m >>= \ (x, s) -> runAWStateT (k x)
instance (GMonad m, GBindable m f f',
          GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(AW, f) '(WO, f') where
    bindG (AWStateT m) k = AWStateT $
        m >>= \ (x, s) -> runWOStateT (k x) >>= \ r ->
            let (y, s') = r in case s' of
                Just z -> return (y, z)
                Nothing -> return (y, s)
instance (GMonad m, GBindable m f f', LeftId m f')
=> GBindable (PermStateT s m) '(AW, f) '(RW, f') where
    bindG (AWStateT m) k = AWStateT $
        m >>= \ (x, s) -> runRWStateT (k x) s

instance (GMonad m, GBindable m f f',
          GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(WO, f) '(Pure, f') where
    bindG (WOStateT m) k = WOSTateT $
        m >>= \ (x, s) -> runPureStateT (k x) >>= \ r -> return (r, s)
instance (GMonad m, GBindable m f f',
          GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(WO, f) '(RO, f') where
    bindG (WOStateT m) k = RWStateT $
        \ s -> m >>= \ r ->
            let (y, s') = r in case s' of
                Just z -> runROStateT (k y) z >>= \ r' -> return (r', z)
                Nothing -> runROStateT (k y) s >>= \ r' -> return (r', s)
instance (GMonad m, GBindable m f f')
=> GBindable (PermStateT s m) '(WO, f) '(AW, f') where
    bindG (WOStateT m) k = AWStateT $
        m >>= \ (x, s) -> runAWStateT (k x)
instance (GMonad m, GBindable m f f',
          GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(WO, f) '(WO, f') where
    bindG (WOStateT m) k = WOSTateT $
        m >>= \ (x, s) -> runWOStateT (k x) >>= \ (y, s') ->
            case s' of
                Just z -> return (y, s')
                Nothing -> return (y, s)
instance (GMonad m, GBindable m f f')
=> GBindable (PermStateT s m) '(WO, f) '(RW, f') where
    bindG (WOStateT m) k = RWStateT $
        \ s -> m >>= \ (x, s') -> case s' of
            Just y -> runRWStateT (k x) y
            Nothing -> runRWStateT (k x) s

```

```

instance (GMonad m, GBindable m f f',
         GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(RW, f) '(Pure, f') where
  bindG (RWStateT f) k = RWStateT $
    \s -> f s >>= \(y, s') -> runPureStateT (k y)
    >>= \r -> return (r, s')

instance (GMonad m, GBindable m f f',
         GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(RW, f) '(RO, f') where
  bindG (RWStateT f) k = RWStateT $
    \s -> f s >>= \(y, s') -> runROStateT (k y) s'
    >>= \r -> return (r, s')

instance (GMonad m, GBindable m f f')
=> GBindable (PermStateT s m) '(RW, f) '(AW, f') where
  bindG (RWStateT f) k = RWStateT $
    \s -> f s >>= \(y, _) -> runAWStateT (k y)

instance (GMonad m, GBindable m f f',
         GBindable m f' (Unit m), LeftId m f')
=> GBindable (PermStateT s m) '(RW, f) '(WO, f') where
  bindG (RWStateT f) k = RWStateT $
    \s -> f s >>= \(x, s') -> runWOStateT (k x)
    >>= \(y, s'') -> case s'' of
      Just z -> return (y, z)
      Nothing -> return (y, s')

instance (GMonad m, GBindable m f f')
=> GBindable (PermStateT s m) '(RW, f) '(RW, f') where
  bindG (RWStateT f) k = RWStateT $
    \s -> f s >>= \(y, s') -> runRWStateT (k y) s'

instance GFunctor m => GFunctor (PermStateT s m) where
  fmapG f (PureStateT m) = PureStateT $ fmapG f m
  fmapG f (ROStateT f') = ROStateT $ \s -> fmapG f (f' s)
  fmapG f (AWStateT m) = AWStateT $
    fmapG (\(x, s) -> (f x, s)) m
  fmapG f (WOStateT m) = WOSTateT $
    fmapG (\(x, s) -> (f x, s)) m
  fmapG f (RWStateT f') = RWStateT $
    \s -> fmapG (\(x, s') -> (f x, s')) (f' s)

instance GMonad m => GMonad (PermStateT s m) where
  type Unit (PermStateT s m) = '(Pure, Unit m)
  type Seq (PermStateT s m) '(e, f) '(e', f') =
    '(PermSeq e e', Seq m f f')
  type Join (PermStateT s m) '(e, f) '(e', f') =
    '(PermJoin e e', Join m f f')
  return = PureStateT . return

instance GMonadTrans (PermStateT s) where
  type IdG (PermStateT s) = Pure
  liftG = PureStateT

getT :: GMonad m => PermStateT s m '(RO, Unit m) s
getT = ROStateT (\s -> return s)

putT :: GMonad m => s -> PermStateT s m '(AW, Unit m) ()

```

```
putT s = AStateT (return ((), s))
```

Maybe

```
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE GADTs               #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE FlexibleContexts    #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE UndecidableInstances #-}

module Control.Granted.Trans.Maybe where

import Prelude hiding (Monad(..))
import Control.Granted
import Control.Granted.Maybe
import Control.Granted.Trans

data GMaybeT m r a where
  GMaybeT :: {runGMaybeT :: m f (GMaybe e a)}
            -> GMaybeT m '(e, f) a

instance (Subeffect m f f')
=> Subeffect (GMaybeT m) '(e, f) '(e, f') where
  sub (GMaybeT x) = GMaybeT (sub x)
instance (Subeffect m f f', Subeffect GMaybe T U, GFunctor m)
=> Subeffect (GMaybeT m) '(T, f) '(U, f') where
  sub (GMaybeT x) = GMaybeT (sub (fmapG sub x))
instance (Subeffect m f f', Subeffect GMaybe F U, GFunctor m)
=> Subeffect (GMaybeT m) '(F, f) '(U, f') where
  sub (GMaybeT x) = GMaybeT (sub (fmapG sub x))

instance (GFunctor GMaybe, GFunctor m)
=> GFunctor (GMaybeT m) where
  fmapG f (GMaybeT x) = GMaybeT (fmapG (fmapG f) x)

type family MaybeTMult (m :: k -> * -> *)
                      (r :: (MaybeGr, k))
                      (s :: (MaybeGr, k)) where
  MaybeTMult m '(F, f) _ = '(F, f)
  MaybeTMult m '(T, f) '(s, f') = '(s, Seq m f f')
  MaybeTMult m '(U, f) '(F, f') = '(F, Seq m f
                                   (Join m f' (Unit m)))
  MaybeTMult m '(U, f) '(_, f') = '(U, Seq m f
                                   (Join m f' (Unit m)))

instance (GBindable GMaybe F e',
          GBindable m f (Unit m), LeftId m f)
=> GBindable (GMaybeT m) '(F, f) '(e', f') where
  bindG (GMaybeT x) k = GMaybeT $ x >> return GNothing
instance (GBindable GMaybe T e', GBindable m f f')
=> GBindable (GMaybeT m) '(T, f) '(e', f') where
  bindG (GMaybeT x) k = GMaybeT $
```

```

x >>= \r -> let GJust y = r in runGMaybeT (k y)

helperUF :: (GMonad m, JoinUpper m f' (Unit m))
=> GMaybe U a -> (a -> GMaybeT m '(F, f') b)
      -> GMaybeT m '(F, Join m f' (Unit m)) b
helperUF (GDyn y) k' = GMaybeT $ case y of
  GNothing -> sub (return GNothing)
  GJust z   -> sub (runGMaybeT (k' z))
instance (GBindable GMaybe U F,
          GBindable m f (Join m f' (Unit m)),
          JoinUpper m f' (Unit m))
=> GBindable (GMaybeT m) '(U, f) '(F, f') where
  bindG (GMaybeT x) k = GMaybeT $
    x >>= \r -> runGMaybeT (helperUF r k)

helperUT :: (GMonad m, JoinUpper m f' (Unit m))
=> GMaybe U a -> (a -> GMaybeT m '(T, f') b)
      -> GMaybeT m '(U, Join m f' (Unit m)) b
helperUT (GDyn y) k' = GMaybeT $ case y of
  GNothing -> sub (return (GDyn GNothing))
  GJust z   -> sub (fmapG GDyn (runGMaybeT (k' z)))
instance (GBindable GMaybe U T,
          GBindable m f (Join m f' (Unit m)),
          JoinUpper m f' (Unit m))
=> GBindable (GMaybeT m) '(U, f) '(T, f') where
  bindG (GMaybeT x) k = GMaybeT $
    x >>= \r -> runGMaybeT (helperUT r k)

helperUU :: (GMonad m, JoinUpper m f' (Unit m))
=> GMaybe U a -> (a -> GMaybeT m '(U, f') b)
      -> GMaybeT m '(U, Join m f' (Unit m)) b
helperUU (GDyn y) k' = GMaybeT $ case y of
  GNothing -> sub (return (GDyn GNothing))
  GJust z   -> sub (runGMaybeT (k' z))
instance (GBindable GMaybe U U,
          GBindable m f (Join m f' (Unit m)),
          JoinUpper m f' (Unit m))
=> GBindable (GMaybeT m) '(U, f) '(U, f') where
  bindG (GMaybeT x) k = GMaybeT $
    x >>= \r -> runGMaybeT (helperUU r k)

instance GMonad m =>
GMonad (GMaybeT m :: (MaybeGr, k) -> * -> *) where
  type Unit (GMaybeT m) = '(T, Unit m)
  type Seq (GMaybeT m) r s = MaybeTMult m r s
  type Join (GMaybeT m) '(e, f) '(e', f') =
    '(MaybeJoin e e', Join m f f')
  return x = GMaybeT (return (GJust x))

instance GMonadTrans GMaybeT where
  type IdG GMaybeT = 'T
  liftG = GMaybeT . (fmapG GJust)

```

C.3 Graded concurrency

These code listings correspond to the graded monad used for concurrency, described in the functional pearl I co-authored with Alan Mycroft [IM20]. The code is so available on the following GitHub repository:

<https://github.com/Dreian/graded-concurrency/>

C.3.1 Graded monad definition

```
{-# LANGUAGE ConstraintKinds      #-}
{-# LANGUAGE DataKinds           #-}
{-# LANGUAGE FlexibleContexts     #-}
{-# LANGUAGE FlexibleInstances    #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE KindSignatures      #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE PolyKinds           #-}
{-# LANGUAGE RebindableSyntax    #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE TypeFamilies        #-}
{-# LANGUAGE TypeOperators       #-}
{-# LANGUAGE UndecidableInstances #-}
```

module Deadlock **where**

```
import           Control.Concurrent
import           Control.Monad.Trans      (lift)
import           Control.Monad.Trans.Reader
import           Data.Functor.Identity
import           Data.Kind
import           Data.Type.Bool
import           Data.Type.Equality
import           GHC.Base                  (liftM)
import           GHC.TypeLits
import           Control.Graded
import           Prelude                  hiding (Monad (..))
import qualified Prelude                  as M
```

-- Synchronized expressions

```
type LockT l m = ReaderT (MVar l) m
```

```
data RegionP = RegionP { pa :: Int, pb :: Int }
data RegionQ = RegionQ { qa :: Int, qs :: String }
instance Show RegionP where
    show reg = "{a=_ " ++ show (pa reg) ++
               ",_b=_ " ++ show (pb reg) ++ "}"
instance Show RegionQ where
    show reg = "{a=_ " ++ show (qa reg) ++
               ",_s=_ " ++ show (qs reg) ++ "}"
```

```
type InternalRep a =
    LockT () (LockT () (LockT () (LockT ()(LockT ()
        (LockT RegionP (LockT RegionQ IO)))))) a
```

```

newtype Sync (lockList :: [Symbol]) a =
  Sync { unSync :: InternalRep a }

infixr 5 :\/:

-- The operations wrap the underlying monad exponentiated by a
-- list of locks / MVars, constructors used to pack and unpack.

instance GBindable Sync e f (e :\/: f) where
  bindG x k = Sync $ x M.>= (unSync . k)

instance Sublist e f => Subeffect Sync e f where
  sub = id

instance GMonad Sync where
  type Unit Sync      = '[]
  type Seq  Sync r s = r :\/: s
  type Join Sync r s = r :\/: s
  alt True x _      = Sync $ unSync x
  alt False _ y     = Sync $ unSync y
  return x = Sync $ M.return x

-- Subtyping relation

class Sublist s t
instance Sublist '[] '[]
instance {-# OVERLAPPABLE #-} Sublist s t => Sublist s (x ': t)
instance {-# OVERLAPS #-} Sublist s t => Sublist (x ': s) (x ': t)

-- List merge operator: :\/:, forces the result to be sorted

type family (xs :: [k]) :\/: (ys :: [k]) :: [k] where
  '[] :\/: ys = ys
  xs :\/: '[] = xs
  (x ': xs) :\/: (y ': ys) =
    If (Cmp x y == LT) (x ': (xs :\/: y ': ys))
    (If (Cmp x y == GT) (y ': (x ': xs :\/: ys))
     (x ': (xs :\/: ys)))

-- 'partial cons' operation for sync functions

class PartialCons (x :: k) (xs :: [k]) (ys :: [k]) | x xs -> ys
-- comment the following line and uncomment the subsequent
-- one to allow deadlock
instance (LessThanAll x xs) => PartialCons x xs (x ': xs)
-- instance PartialCons x xs (x ': xs)

-- Helper operators for comparison and testing

type family Cmp (a :: k) (b :: k) :: Ordering
type instance Cmp (v :: Symbol) (u :: Symbol) = CmpSymbol v u

class BoolAsConstraint b
instance BoolAsConstraint True

```

```

class LessThanAll (x :: k) (ys :: [k])
instance LessThanAll x '[]
instance BoolAsConstraint (Cmp x y == LT)
=> LessThanAll x (y ': ys)

type LockListProperties r s = ((r :\/: r) ~ r, (s :\/: s) ~ s,
                               (r :\/: s) ~ (s :\/: r),
                               ((r :\/: s) :\/: r) ~ (r :\/: s),
                               ((r :\/: s) :\/: s) ~ (r :\/: s),
                               Sublist r (r :\/: s),
                               Sublist '[] r, Sublist '[] s,
                               (r :\/: '[]) ~ r, ('[] :\/: r) ~ r,
                               (s :\/: '[]) ~ s, ('[] :\/: s) ~ s)

-- Run the monad

performSync :: Sync r a -> MVar () -> MVar () -> MVar ()
              -> MVar () -> MVar ()
              -> MVar RegionP -> MVar RegionQ -> IO a
performSync m lv lw lx ly lz lp lq =
  runReaderT (runReaderT (runReaderT (runReaderT (runReaderT
    (runReaderT (runReaderT (unSync m) lv) lw) lx) ly) lz) lp) lq

runSync :: Sync r a -> () -> () -> () -> () -> ()
         -> RegionP -> RegionQ -> IO a
runSync m v w x y z p q =
  newMVar v M.>>= \lv -> newMVar w M.>>= \lw ->
  newMVar x M.>>= \lx -> newMVar y M.>>= \ly ->
  newMVar z M.>>= \lz -> newMVar p M.>>= \lp ->
  newMVar q M.>>= \lq -> performSync m lv lw lx ly lz lp lq

-- Run in parallel

(<||>) :: Sync r a -> Sync s b -> Sync (r :\/: s) (a, b)
Sync p <||> Sync q = do
  la <- liftIO newEmptyMVar
  lb <- liftIO newEmptyMVar
  Sync $ unSync $ deepFork (Sync $ bindIO (putMVar la) p)
                           (Sync $ bindIO (putMVar lb) q)
  x <- liftIO $ takeMVar la
  y <- liftIO $ takeMVar lb
  return (x, y)

-- Synchronization, locking functions

makeSync :: Sync '[] (MVar l) -> (l -> Sync r (a, l)) -> Sync s a
makeSync l k = Sync $ unSync $ do
  lock <- l
  v <- liftIO (takeMVar lock)
  (res, newv) <- k v
  liftIO (putMVar lock newv)
  return res

ignoreLock :: Sync r a -> l -> Sync r (a, l)
ignoreLock m l = do
  v <- m

```

```

    return (v, l)

syncP :: PartialCons "P" r s
=> (RegionP -> Sync r (a, RegionP)) -> Sync s a
syncP = makeSync $ Sync $ lift $ lift $ lift $ lift $ lift ask

syncQ :: PartialCons "Q" r s
=> (RegionQ -> Sync r (a, RegionQ)) -> Sync s a
syncQ = makeSync $ Sync $ lift $ lift $ lift $ lift $ lift $
    lift ask

syncV :: PartialCons "V" r s => Sync r a -> Sync s a
syncV m = makeSync (Sync ask) $ ignoreLock m

syncW :: PartialCons "W" r s => Sync r a -> Sync s a
syncW m = makeSync (Sync $ lift ask) $ ignoreLock m

syncX :: PartialCons "X" r s => Sync r a -> Sync s a
syncX m = makeSync (Sync $ lift $ lift ask) $ ignoreLock m

syncY :: PartialCons "Y" r s => Sync r a -> Sync s a
syncY m = makeSync (Sync $ lift $ lift $ lift ask) $ ignoreLock m

syncZ :: PartialCons "Z" r s => Sync r a -> Sync s a
syncZ m = makeSync (Sync $ lift $ lift $ lift $ lift ask) $
    ignoreLock m

-- IO operations as Sync operations

pureReaderT :: M.Monad m => m a -> ReaderT s m a
pureReaderT = ReaderT . M.return

pushBind :: M.Monad m => (a -> m b) -> LockT s m a -> LockT s m b
pushBind k m = ReaderT $ \x -> runReaderT m x M.>= k

pushDeepBind :: (M.Monad m, M.Monad n) =>
((a -> n b) -> m a -> m b) -> (a -> n b) -> LockT r m a
    -> LockT r m b
pushDeepBind l k m = ReaderT $ \x -> l k $ runReaderT m x

deepFork :: Sync r () -> Sync s () -> Sync '[] ThreadId
deepFork m n = Sync $ ReaderT $ \v -> ReaderT $ \w ->
    ReaderT $ \x -> ReaderT $ \y ->
    ReaderT $ \z -> ReaderT $ \p ->
    ReaderT $ \q ->
    forkIO (performSync m v w x y z p q) M.>>
    forkIO (performSync n v w x y z p q)

bindIO :: (a -> IO b) -> InternalRep a -> InternalRep b
bindIO = pushDeepBind $ pushDeepBind $ pushDeepBind $
    pushDeepBind $ pushDeepBind $ pushDeepBind pushBind

liftIOInternal :: IO a -> InternalRep a
liftIOInternal = pureReaderT . pureReaderT . pureReaderT .
    pureReaderT . pureReaderT . pureReaderT .

```

```

        pureReaderT

liftIO :: IO a -> Sync '[] a
liftIO = Sync . liftIOInternal

delay :: Int -> Sync '[] ()
delay = liftIO . threadDelay

syncPutStrLn :: String -> Sync '[] ()
syncPutStrLn = liftIO . putStrLn

-- Helpful function for sequencing based on a list

foldBind :: (Foldable f, LockListProperties r s)
          => Sync r a -> f (a -> Sync s a) -> Sync (r :\/: s) a
foldBind m = foldl (>=) (sub m)

repeatBind :: LockListProperties r r =>
Int -> Sync r a -> Sync r a
repeatBind n m = m `foldBind` replicate (n - 1) (const m)

```

C.3.2 Usage examples

```

{-# LANGUAGE FlexibleContexts #-}
{-# LANGUAGE RebindableSyntax #-}
{-# LANGUAGE TypeFamilies      #-}

import           Prelude      hiding (Monad, return, (>>), (>=))
import qualified Prelude      as M

import           Control.Graded
import           Deadlock

-- Dining philosophers: demo

philosopher x s1 s2 = s1 $ do
    delay 100000
    s2 $ do
        delay 100000
        syncPutStrLn $ "philosopher_" ++ show x

philosopher1 = philosopher 1 syncV syncW
philosopher2 = philosopher 2 syncW syncX
philosopher3 = philosopher 3 syncX syncY
philosopher4 = philosopher 4 syncY syncZ
philosopher5 = philosopher 5 syncV syncZ

dining = philosopher1 <||> philosopher2 <||> philosopher3
        <||> philosopher4 <||> philosopher5

repeatEat p = repeatBind 5 p

longDining = repeatEat philosopher1 <||> repeatEat philosopher2
            <||> repeatEat philosopher3 <||> repeatEat philosopher4
            <||> repeatEat philosopher5

```

```

-- Dining philosophers: examples of expressions that do not type
check
-- Uncomment the definitions of philosopher3' and dining'

-- philosopher5' = philosopher 5 syncZ syncV

-- dining' = philosopher1 <||> philosopher2 <||> philosopher3
--           <||> philosopher4 <||> philosopher5'

-- Showing concurrency in action

threads = runSync (syncPutStrLn "[thread_1]" <||>
                    syncPutStrLn "[thread_2]")
              () () () () () (RegionP 0 0) (RegionQ 0 "")
threadsIsolated = runSync (syncX (syncPutStrLn "[thread_1]" <||>
                                   syncX (syncPutStrLn "[thread_2]"))
                           () () () () () (RegionP 0 0) (RegionQ 0 ""))

-- Memory regions example

thr1 = do
  delay 1000000
  syncP $ \p -> syncQ $ \q ->
    return ((0, p), RegionQ (pa p + 1) "touched_by_1")
thr2 = do
  delay 1000000
  syncP $ \p ->
    return (1, RegionP 50 10)
thr3 = do
  delay 1000000
  syncQ $ \q ->
    return (2, RegionQ (qa q + 5) "touched_by_3")
thr4 = do
  delay 1000000
  syncP $ \p -> syncQ $ \q ->
    return ((0, RegionP (2 * qa q + 1) (pa p)),
            RegionQ (qa q) "touched_by_4")

regions = do
  thr1 <||> thr2 <||> thr3 <||> thr4
  syncQ $ \q -> do
    syncPutStrLn $ show q
    return ((), q)
  return "done"

regionsRun = runSync regions
              () () () () () (RegionP 0 0) (RegionQ 0 "")

-- Run dining philosophers

main :: IO ()

```

```
main = runSync dining () () () () () (RegionP 0 0) (RegionQ 0 "")  
      M.>> putStrLn "Done!"
```


Denotational semantics of System G

D.1 Overview of denotational semantics

We first provide the *definitions* of semantics of contexts; the denotations of judgements are merely *specified* by giving their *semantic domains* and their exact treatment is deferred to the following section.

We define the remainder of these denotations in a largely causal order: each definition depends on the previous ones in the sequence. However, we start with the semantics of the typing judgements for value-kinded terms in a given context. Three similar cases, denotations of contexts, are listed together at the end.

Denotation of value-kinded typing judgements The semantics of a typing judgement $\llbracket \mathcal{J} \vdash_{\text{val}} M : A \rrbracket_{\text{val}}$ is a map from a context record $\mathcal{R}$ to some member of the semantics of the type A : $\llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R}_{\Sigma, \mathcal{S}, \Delta, \Phi}$. Here $\mathcal{R}$ is a record over ‘ Σ ’, ‘ $\mathcal{S}$ ’, ‘ Δ ’, ‘ Φ ’ and ‘ Γ ’, which respectively contain a set of symbols, a map from symbol set variable names to finite sets of symbols, a map from symbol variable names to symbol constants, a map from grade variable names to elements of some grading algebras, and a map from variable names to elements of the semantics of their types (domain of $\mathcal{R}_{\Gamma}$ is $\llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{val2type}} \Gamma \rrbracket^{\text{val2type}} \mathcal{R}_{\Sigma, \mathcal{S}, \Delta, \Phi}$).

We do not elaborate on the components of the records $\mathcal{R}$ in each of the following explanations, since they follow the ones for typing judgements. The denotations of contexts and judgements are:

Symbol-constant contexts $\llbracket \Sigma \rrbracket^{\text{symcenv}}$ is trivial, just giving the set of elements included in Σ , e.g. $\llbracket X, Y \rrbracket^{\text{symcenv}} = \{X, Y\}$.

Symbol-set-variables context $\llbracket \mathcal{S} \rrbracket^{\text{ssenv}}$ plays no role at run-time (even if $\mathcal{S}$ is necessary for entailments) and thus can be interpreted as just a unit (singleton set). This differs from what $\mathcal{R}_{\mathcal{S}}$ is (map from symbol set variable names to sets of symbols).

Symbol set judgements The denotation $\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}}$ of symbol set S under the context $\Sigma \mid \mathcal{S}$ is a function mapping records $\mathcal{R}$ with fields ‘ Σ ’ and ‘ $\mathcal{S}$ ’, to some finite set whose elements are drawn from $\llbracket \Sigma \rrbracket^{\text{symcenv}}$ and the range of $\mathcal{R}_{\mathcal{S}}$ (Appendix).

Symbol judgements The denotation $\llbracket \Sigma \mid \mathcal{S} \mid \Delta \vdash_{\text{sym}} \Psi \in S \rrbracket_{\text{sym}}$ of symbol Ψ in context $\Sigma \mid \mathcal{S} \mid \Delta$ is a function from a record $\mathcal{R}$ with fields ‘ Σ ’, $\mathcal{S}$ and ‘ Δ ’ (as before) to an element of $\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}$ (Appendix).

Graded monad builders The semantics of each graded monad builder T , $\llbracket T \rrbracket^{\text{gmb}}$, is a function mapping a finite sets of symbols ζ to a grading algebra. Constructing types, the other meaning of graded monad builders, is part of the semantics of types. For three interesting graded monad builders, **Exception**, **StPerm** and **StLive**, these definitions are given in Figure D.1.

Grading algebras A grading algebra, which we refer to $[T]S$ in syntax (for example, in grade abstractions), corresponds semantically to a computational grading algebra denoted by $T^\circ S$. The semantics of a grading algebra $[T]S$ (for a graded monad with underlying graded monad builder T and symbol set S), $\llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket^{\text{alg}}$, is a function that maps records $\mathcal{R}$ with fields ' Σ ', ' S ' and ' Δ ' (as before) to the grading algebra associated with this graded monad (the set S , naturally, being interpreted as $\llbracket \Sigma \mid S \vdash_{\text{ss}} S \rrbracket_{\text{ss}}$). The definition is given by the equation:

$$\llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket^{\text{alg}} \mathcal{R} = \llbracket T \rrbracket^{\text{gmb}} (\llbracket \Sigma \mid S \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R})$$

The resulting grading algebra of $\llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket^{\text{alg}} \mathcal{R}$ is a tuple $(E, \sqsubseteq, \odot, \mathbf{1}, O)$. The individual components of this are accessed using dictionary notation, e.g. $(\llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket^{\text{alg}} \mathcal{R})[\sqsubseteq]$.

Grade judgements $\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \varphi \rrbracket_{\text{gr}}$ is a function from a record $\mathcal{R}$ with fields Σ , S , Δ and Φ (first three as before, and Φ is a map from grade variable names to elements of various grading algebras) to the semantics of grade-kinded term F , associated with grading algebra $\llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket^{\text{alg}} \mathcal{R}_{\Sigma, S, \Delta}$. The semantics of F is either the corresponding element of the grading algebra or a grade-kinded operator. (Details in the following section.)

Type judgements $\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}}$ is a map from a record $\mathcal{R}$ with fields Σ , S , Δ and Φ (all as before) to the semantics of A , which is a domain. (Details in the following section.)

Value-variable type contexts We lift the $\vdash_{\text{type}}$ judgements from types to type environments conjunctively to give $\vdash_{\text{val2type}}$ defined by $\Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{val2type}} \Gamma$ which holds exactly when $\Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} \Gamma(x)$ holds for all $x \in \text{dom}(\Gamma)$. This judgement form has semantics:

$$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{val2type}} \Gamma \rrbracket^{\text{val2type}} \mathcal{R} = \prod_{x \in \text{dom}(\Gamma)} \llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} \Gamma(x) \rrbracket_{\text{type}} \mathcal{R}$$

Symbol-variable contexts Analogously to value-variables, we lift $\llbracket \cdot \rrbracket_{\text{ss}}$ to obtain $\llbracket \cdot \rrbracket^{\text{sym2ss}}$:

$$\llbracket \Sigma \mid S \vdash_{\text{sym2ss}} \Delta \rrbracket^{\text{sym2ss}} \mathcal{R} = \prod_{\delta \in \text{dom}(\Delta)} \llbracket \Sigma \mid S \vdash_{\text{ss}} \Delta(\delta) \rrbracket_{\text{ss}} \mathcal{R}$$

Note that $\mathcal{R}$ here only has Σ, S components.

Grade-variable contexts Again, we similarly lift $\llbracket \cdot \rrbracket^{\text{alg}}$ to obtain $\llbracket \cdot \rrbracket^{\text{gr2alg}}$:

$$\llbracket \Sigma \mid S \mid \Delta \vdash_{\text{gr2alg}} \Phi \rrbracket^{\text{gr2alg}} \mathcal{R} = \prod_{r \in \text{dom}(\Phi)} \llbracket \Sigma \mid S \vdash_{\text{alg}} \Phi(r) \rrbracket^{\text{alg}} \mathcal{R}_{\Sigma, S}$$

or, in other words, a function from context record $\mathcal{R}$ to a partial map from grade variable names to elements of the grading algebras they belong to.

$$\begin{aligned}
\llbracket \text{Exception} \rrbracket^{\text{gmb}} \zeta &= (\mathcal{P}(\zeta), \subseteq, \cup, \emptyset, \bigcup_{V \in \zeta} \{\text{add } V, \text{rem } V\}) \\
\llbracket \text{StPerm} \rrbracket^{\text{gmb}} \zeta &= (\zeta \rightarrow \{\text{pure}, \text{ro}, \text{wo}, \text{rw}\}, \sqsubseteq, \sqcup, \lambda x \in \zeta. \text{pure}, \bigcup_{V \in \zeta} \{\text{rd } V, \text{wr } V\}) \\
\llbracket \text{StLive} \rrbracket^{\text{gmb}} \zeta &= (\mathcal{P}(\zeta) \rightarrow \mathcal{P}(\zeta), \sqsubseteq, \cup, \lambda s \in \mathcal{P}(\zeta). \emptyset, \bigcup_{V \in \zeta} \{\text{gen } V, \text{kill } V\})
\end{aligned}$$

Figure D.1. Interpreting graded monad builders when applied to finite set of symbols ζ (for $T^\circ S$).

$$\begin{aligned}
\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} \{\{X_1, \dots, X_k\}\} \rrbracket_{\text{ss}} \mathcal{R} &= \{X_i \mid 1 \leq i \leq k\} \\
\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} \xi \rrbracket_{\text{ss}} \mathcal{R} &= \mathcal{R}_{\mathcal{S}}(\xi) \\
\llbracket \Sigma \mid \mathcal{S} \mid \Delta \vdash_{\text{sym}} X \in S \rrbracket_{\text{sym}} \mathcal{R} &= X \\
\llbracket \Sigma \mid \mathcal{S} \mid \Delta \vdash_{\text{sym}} \delta \in S \rrbracket_{\text{sym}} \mathcal{R} &= \mathcal{R}_{\Delta}(\delta)
\end{aligned}$$

Figure D.2. Denotational semantics of symbol sets and symbols.

Above we have defined the semantics for each of the five contexts, but only specified the semantic domains of the six kinds of entailments $\llbracket \cdot \rrbracket_{\text{val}}, \llbracket \cdot \rrbracket_{\text{type}}, \llbracket \cdot \rrbracket_{\text{gr}}, \llbracket \cdot \rrbracket_{\text{ss}}, \llbracket \cdot \rrbracket_{\text{sym}}$; their definitions are given in the Appendix. The one definition we give in this section is the semantics of grading algebras – what mathematical object $T^\circ S$ corresponds to.

D.2 System G: full denotational semantics

First we define the semantics of symbols and symbol sets. A symbol is either a symbol variable or a symbol constant, and thus the denotation of a symbol is a function from a context map $\mathcal{R}$ to either the value of a symbol variable or a constant. Symbol sets are similar and are either symbol-set variables or symbol-set constants (so they are interpreted either as sets of constant symbols or as sets seen in the range of $\mathcal{R}_{\mathcal{S}}$). The denotational semantics of symbols and symbol sets is given in Figure D.2.

Grade-level term semantics is, for the most part, easy to define. The denotation of a grade in context is a function from a record $\mathcal{R}$ to an element of the corresponding

$$\begin{aligned}
\llbracket \mathcal{J} \vdash_{\text{gr}} 1 \triangleleft [T]S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} &= \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{alg}} T^\circ S \rrbracket^{\text{alg}} \mathcal{R}_{\Sigma, \mathcal{S}}[1] \\
\llbracket \mathcal{J} \vdash_{\text{gr}} r \triangleleft [T]S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} &= \mathcal{R}_{\Phi}(r) \\
\llbracket \mathcal{J} \vdash_{\text{gr}} F \odot G \triangleleft [T]S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} &= \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{alg}} T^\circ S \rrbracket^{\text{alg}} \mathcal{R}_{\Sigma, \mathcal{S}}[\odot] \\
&\quad (\llbracket \mathcal{J} \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R}, \\
&\quad \llbracket \mathcal{J} \vdash_{\text{gr}} G \triangleleft [T]S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R}) \\
\llbracket \mathcal{J} \vdash_{\text{gr}} O \triangleleft [T]S \ \& \ \varphi \rrbracket_{\text{gr}} \mathcal{R} &= \text{depending on } T \\
\llbracket \mathcal{J} \vdash_{\text{gr}} F \ G \triangleleft [T]S \ \& \ \varphi \rrbracket_{\text{gr}} \mathcal{R} &= (\llbracket \mathcal{J} \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Eff} \Rightarrow \varphi \rrbracket_{\text{gr}} \mathcal{R}) \\
&\quad (\llbracket \mathcal{J} \vdash_{\text{gr}} G \triangleleft [T]S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R}) \\
\llbracket \mathcal{J} \vdash_{\text{gr}} F \ \Psi \triangleleft [T]S \ \& \ \varphi \rrbracket_{\text{gr}} \mathcal{R} &= (\llbracket \mathcal{J} \vdash_{\text{gr}} F \triangleleft [T]S \ \& \ \text{Sym} \Rightarrow \varphi \rrbracket_{\text{gr}} \mathcal{R}) \\
&\quad (\llbracket \Sigma \mid \mathcal{S} \mid \Delta \vdash_{\text{sym}} \Psi \in S \rrbracket_{\text{sym}} \mathcal{R}_{\Sigma, \mathcal{S}, \Delta})
\end{aligned}$$

Figure D.3. The denotational semantics of effects. Here $\mathcal{J}$ abbreviates $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi$.

$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A_1 \rightarrow A_2 \rrbracket_{\text{type}} \mathcal{R}$	$= (\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A_1 \rrbracket_{\text{type}} \mathcal{R})$ $\rightarrow (\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A_2 \rrbracket_{\text{type}} \mathcal{R})$
$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} \forall \text{eff } r : [T] S (A) \rrbracket_{\text{type}} \mathcal{R}$	$= (\llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket_{\text{alg}}^{\text{alg}} \mathcal{R}_{\Sigma, S})$ $\rightarrow (\llbracket \Sigma \mid S \mid \Delta \mid r : [T] S, \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R} \{ \Phi \mapsto \mathcal{R}_\Phi [r \mapsto \llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket_{\text{alg}}^{\text{alg}} \mathcal{R}_{\Sigma, S} [T] S] \})$
$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} \Pi_{\text{sym}} \delta : S (A) \rrbracket_{\text{type}} \mathcal{R}$	$= (\llbracket \Sigma \mid S \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S})$ $\rightarrow (\llbracket \Sigma \mid S \mid \delta : S, \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R} \{ \Delta \mapsto \mathcal{R}_\Delta [\delta \mapsto S] \})$
$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} \Pi_{\text{ss}} \xi (A) \rrbracket_{\text{type}} \mathcal{R} \zeta$	$= \llbracket \Sigma \mid \xi, S \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R} \{ S \mapsto \mathcal{R}_S [\xi \mapsto \zeta] \}$
$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A F \rrbracket_{\text{type}} \mathcal{R}$	$= (\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R})$ $(\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{gr}} F \triangleleft T^\circ S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R})$
$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A \Psi \rrbracket_{\text{type}} \mathcal{R}$	$= (\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R})$ $(\llbracket \Sigma \mid S \mid \Delta \vdash_{\text{sym}} \Psi \in S \rrbracket_{\text{sym}} \mathcal{R}_{\Sigma, S, \Delta})$
$\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A S \rrbracket_{\text{type}} \mathcal{R}$	$= (\llbracket \Sigma \mid S \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type}} \mathcal{R})$ $(\llbracket \Sigma \mid S \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S})$

Figure D.4. Interpreting types which are not graded monadic.

$\llbracket \mathcal{I} \vdash_{\text{val}} x : A \rrbracket_{\text{val}} \mathcal{R}$	$= \mathcal{R}_\Gamma(x)$
$\llbracket \mathcal{I} \vdash_{\text{val}} \nu X (M) : A \rrbracket_{\text{val}} \mathcal{R}$	$= \llbracket \mathcal{I} \{ \Sigma \mapsto X, \mathcal{I}_\Sigma \} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} \mathcal{R} \{ \Sigma \mapsto \{X\} \cup \mathcal{R}_\Sigma \}$
$\llbracket \mathcal{I} \vdash_{\text{val}} \lambda x : B (M) : A \rrbracket_{\text{val}} \mathcal{R}$	$= \lambda t \in \llbracket \mathcal{I}_{\Sigma, S, \Delta, \Phi} \vdash_{\text{type}} B \rrbracket_{\text{type}}.$ $(\llbracket \mathcal{I} \{ \Gamma \mapsto x : B, \mathcal{I}_\Gamma \} \vdash_{\text{val}} M \rrbracket_{\text{val}} \mathcal{R} \{ \Gamma \mapsto \mathcal{R}_\Gamma [x \mapsto t] \})$
$\llbracket \mathcal{I} \vdash_{\text{val}} M N : A \rrbracket_{\text{val}} \mathcal{R}$	$= (\llbracket \mathcal{I} \vdash_{\text{val}} M : B \rightarrow A \rrbracket_{\text{val}} \mathcal{R}) (\llbracket \mathcal{I} \vdash_{\text{val}} N : B \rrbracket_{\text{val}} \mathcal{R})$
$\llbracket \mathcal{I} \vdash_{\text{val}} \Lambda \text{eff } r : [T] S (M) : A \rrbracket_{\text{val}} \mathcal{R}$	$= \lambda \phi \in \llbracket \mathcal{I}_{\Sigma, S} \vdash_{\text{alg}} T^\circ S \rrbracket_{\text{alg}}^{\text{alg}} \mathcal{R}_{\Sigma, S, \Delta}.$ $\llbracket \mathcal{I} \{ \Phi \mapsto r : [T] S, \mathcal{I}_\Phi \} \vdash_{\text{val}} M \rrbracket_{\text{val}} \mathcal{R} \{ \Phi \mapsto \mathcal{R}_\Phi [r \mapsto [T] S] \}$
$\llbracket \mathcal{I} \vdash_{\text{val}} M F : A \rrbracket_{\text{val}} \mathcal{R}$	$= (\llbracket \mathcal{I} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} \mathcal{R}) (\llbracket \mathcal{I}_{\Sigma, S, \Delta, \Phi} \vdash_{\text{gr}} F \triangleleft T^\circ S \ \& \ \text{Eff} \rrbracket_{\text{gr}} \mathcal{R}_{\Sigma, S, \Delta, \Phi})$
$\llbracket \mathcal{I} \vdash_{\text{val}} \lambda_{\text{sym}} \delta : S (M) : A \rrbracket_{\text{val}} \mathcal{R}$	$= \lambda \psi \in \llbracket \mathcal{I}_{\Sigma, S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}.$ $(\llbracket \mathcal{I} \{ \Delta \mapsto \delta : S, \mathcal{I}_\Delta \} \vdash_{\text{val}} M \rrbracket_{\text{val}} \mathcal{R} \{ \Delta \mapsto \mathcal{R}_\Delta [\delta \mapsto \psi] \})$
$\llbracket \mathcal{I} \vdash_{\text{val}} M \Psi : A \rrbracket_{\text{val}} \mathcal{R}$	$= (\llbracket \mathcal{I} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} \mathcal{R})$ $(\llbracket \mathcal{I}_{\Sigma, S, \Delta} \vdash_{\text{sym}} \Psi \in S \rrbracket_{\text{sym}} \mathcal{R}_{\Sigma, S, \Delta})$
$\llbracket \mathcal{I} \vdash_{\text{val}} \lambda_{\text{ss}} \xi (M) : A \rrbracket_{\text{val}} \mathcal{R} \zeta$	$= (\llbracket \mathcal{I} \{ S \mapsto \xi, \mathcal{I}_S \} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} \mathcal{R}) (\mathcal{R} \{ S \mapsto \mathcal{R}_S [\xi \mapsto \zeta] \})$
$\llbracket \mathcal{I} \vdash_{\text{val}} M S : A \rrbracket_{\text{val}} \mathcal{R}$	$= (\llbracket \mathcal{I} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} \mathcal{R}) (\llbracket \mathcal{I}_{\Sigma, S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S})$

Figure D.5. Interpreting value-kinded terms that do not concern graded monads.

algebra or a grade-level function. The denotational semantics of the identity grade **1** is a function from the record $\mathcal{R}$ to the denotation of the identity of the corresponding grading algebra – and the denotation of the grading algebra is given by applying the denotation of $\llbracket \Sigma \mid S \vdash_{\text{alg}} T^\circ S \rrbracket_{\text{alg}}^{\text{alg}}$ to the sub-record $\mathcal{R}_{\Sigma, S}$. Similarly, the denotation of a product of two grades (that is, $F \odot G$) is a function from a record to the application of the denotation of the grade multiplication operator in the appropriate grading algebra to the denotations of F and G . The other cases are similar, with special attention paid to the grade-level constants and operators – these are defined later, when we look at the semantics of graded monadic operations for the three interesting graded monads (**Exception**, **StPerm**, **StLive**). Overall the semantics of grade-level terms is given in Figure D.3.

The denotational semantics of types is given in Figure D.4. The definition follows the standard ideas from denotational semantics of simply-typed lambda calculus, PCF and similar languages. The figure only shows the semantics of types that are not graded monadic – the semantics of those types has to be defined separately. Grade, symbol and symbol-set abstractions (polymorphism) have as denotations maps from $\mathcal{R}$ to denotations of all possible results of application – this is in the form of set of all pairs of denotations of effects/symbols/symbol sets and denotations of the result of application.

$$\begin{aligned}
\llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} \text{Exception}^\diamond S F A \rrbracket_{\text{type} \mathcal{R}} &= \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type} \mathcal{R}} + \\
&\quad \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss} \mathcal{R}_{\Sigma, \mathcal{S}}} \\
\llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} \text{StPerm}^\diamond S F A \rrbracket_{\text{type} \mathcal{R}} &= (\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss} \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z}) \rightarrow \\
&\quad \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type} \mathcal{R}} \times \\
&\quad (\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss} \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z}) \\
\llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} \text{StLive}^\diamond S F A \rrbracket_{\text{type} \mathcal{R}} &= (\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss} \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z}) \rightarrow \\
&\quad \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{type}} A \rrbracket_{\text{type} \mathcal{R}} \times \\
&\quad (\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss} \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z})
\end{aligned}$$

Figure D.6. Interpreting graded monadic types. These are simplified: rather than having different representations and semantics depending on grade F , the grade merely provides simple analysis information.

Next, the denotations of value terms are stated in Figure D.5, except for the monadic bind, return and subeffecting. No part of the denotational semantics is particularly surprising.

Graded monad builders are associated with maps from sets to grading algebras parameterised by these sets. The denotations are given in Figure D.1 for the three kinds of graded monads we consider (**Exception**, **StPerm**, **StLive**). Furthermore, a graded monadic type $T^\diamond S F A$ has a denotation that depends on T – the denotational semantics of these types is given in Figure D.6.

Next, we need to state the denotational semantics of value-level terms for bind, return and subeffecting for all three graded monad builders. The semantics of these operations is stated in Figure D.7. Note that the semantics of $\text{sub}_{T^\diamond} F G M$ is simple: the grade does not change the underlying representation of the graded monad.

Finally, we define the semantics of all graded monad operations and effect algebra operators – essentially value-level and effect-level constants. They follow the common implementations of these operations and representations of the graded monads in programming languages. The remainder of the semantics is given in Figures D.8 and D.9.

$\llbracket \mathcal{J} \vdash_{\text{val}} \text{ret}_{\text{Exception}}^{\star_S} M : \text{Exception}^{\star_S} \mathbf{1} A \rrbracket_{\text{val}, \mathcal{R}}$	$= \text{inl} (\llbracket \mathcal{J} \vdash_{\text{val}} M : A \rrbracket_{\text{val}})$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{ret}_{\text{StPerm}}^{\star_S} M : \text{StPerm}^{\star_S} \mathbf{1} A \rrbracket_{\text{val}, \mathcal{R}}$	$= \lambda s \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}, \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z}.$ $(\llbracket \mathcal{J} \vdash_{\text{val}} M : A \rrbracket_{\text{val}, s})$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{ret}_{\text{StLive}}^{\star_S} M : \text{StLive}^{\star_S} \mathbf{1} A \rrbracket_{\text{val}, \mathcal{R}}$	$= \lambda s \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}, \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z}.$ $(\llbracket \mathcal{J} \vdash_{\text{val}} M : A \rrbracket_{\text{val}, s})$
$\llbracket \mathcal{J} \vdash_{\text{val}} M \gg_{\text{Exception}}^{\star_S} N : \text{Exception}^{\star_S} (F \odot G) B \rrbracket_{\text{val}, \mathcal{R}}$	$= \begin{cases} (\llbracket \mathcal{J} \vdash_{\text{val}} N : A \rightarrow \text{Exception}^{\star_S} G B \rrbracket_{\text{val}, \mathcal{R}})(x), \\ \text{if } \llbracket \mathcal{J} \vdash_{\text{val}} M : \text{Exception}^{\star_S} F A \rrbracket_{\text{val}, \mathcal{R}} = \text{inl} x \\ \llbracket \mathcal{J} \vdash_{\text{val}} M : \text{Exception}^{\star_S} F A \rrbracket_{\text{val}, \mathcal{R}}, \\ \text{otherwise} \end{cases}$
$\llbracket \mathcal{J} \vdash_{\text{val}} M \gg_{\text{StPerm}}^{\star_S} N : \text{StPerm}^{\star_S} (F \odot G) B \rrbracket_{\text{val}, \mathcal{R}}$	$= \lambda s \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}, \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z}.$ $\llbracket \mathcal{J} \vdash_{\text{val}} N : A \rightarrow \text{StPerm}^{\star_S} G B \rrbracket_{\text{val}}$ $(\text{fst} (\llbracket \mathcal{J} \vdash_{\text{val}} M : \text{StPerm}^{\star_S} F A \rrbracket_{\text{val}, s}))$ $(\text{snd} (\llbracket \mathcal{J} \vdash_{\text{val}} M : \text{StPerm}^{\star_S} F A \rrbracket_{\text{val}, s}))$
$\llbracket \mathcal{J} \vdash_{\text{val}} M \gg_{\text{StLive}}^{\star_S} N : \text{StLive}^{\star_S} (F \odot G) B \rrbracket_{\text{val}, \mathcal{R}}$	$= \lambda s \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}, \mathcal{R}_{\Sigma, \mathcal{S}}} \rightarrow \mathbb{Z}.$ $\llbracket \mathcal{J} \vdash_{\text{val}} N : A \rightarrow \text{StLive}^{\star_S} G B \rrbracket_{\text{val}}$ $(\text{fst} (\llbracket \mathcal{J} \vdash_{\text{val}} M : \text{StLive}^{\star_S} F A \rrbracket_{\text{val}, s}))$ $(\text{snd} (\llbracket \mathcal{J} \vdash_{\text{val}} M : \text{StLive}^{\star_S} F A \rrbracket_{\text{val}, s}))$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{sub}_{\text{Exception}}^{\star_S} F G M : \text{Exception}^{\star_S} G A \rrbracket_{\text{val}, \mathcal{R}}$	$= \llbracket \mathcal{J} \vdash_{\text{val}} M : \text{Exception}^{\star_S} F A \rrbracket_{\text{val}, \mathcal{R}}$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{sub}_{\text{StPerm}}^{\star_S} F G M : \text{Exception}^{\star_S} G A \rrbracket_{\text{val}, \mathcal{R}}$	$= \llbracket \mathcal{J} \vdash_{\text{val}} M : \text{StPerm}^{\star_S} F A \rrbracket_{\text{val}, \mathcal{R}}$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{sub}_{\text{StLive}}^{\star_S} F G M : \text{Exception}^{\star_S} G A \rrbracket_{\text{val}, \mathcal{R}}$	$= \llbracket \mathcal{J} \vdash_{\text{val}} M : \text{StLive}^{\star_S} F A \rrbracket_{\text{val}, \mathcal{R}}$

(Here $\mathcal{J}$ abbreviates $\Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \mid \Gamma$)

Figure D.7. Semantics of graded monadic operations: return, bind and subeffecting.

getPerm	$: \Pi_{\text{ss}} \xi (\Pi_{\text{sym}} \delta : \xi (\text{StPerm}^{\star_S} \xi (\text{rdPerm } \delta) \text{Int}))$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{getPerm} : C \rrbracket_{\text{val}, \mathcal{R} \zeta}$	$= \lambda \psi \in \zeta. \lambda s \in \mathbb{Z}. (s(\psi), s)$
putPerm	$: \Pi_{\text{ss}} \xi (\Pi_{\text{sym}} \delta : \xi (\text{Int} \rightarrow \text{StPerm}^{\star_S} \xi (\text{wrPerm } \delta) \text{Unit}))$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{putPerm} : C \rrbracket_{\text{val}, \mathcal{R} \zeta}$	$= \lambda \psi \in \zeta. \lambda v \in \mathbb{Z}. \lambda s \in \mathbb{Z}. ((\text{Int}, s + \{\xi \mapsto v\}))$
getLive	$: \Pi_{\text{ss}} \xi (\Pi_{\text{sym}} \delta : \xi (\text{StLive}^{\star_S} \xi (\text{rdLive } \delta) \text{Int}))$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{getLive} : C \rrbracket_{\text{val}, \mathcal{R} \zeta}$	$= \lambda \psi \in \zeta. \lambda s \in \mathbb{Z}. (s(\psi), s)$
putLive	$: \Pi_{\text{ss}} \xi (\Pi_{\text{sym}} \delta : \xi (\text{Int} \rightarrow \text{StLive}^{\star_S} \xi (\text{wrLive } \delta) \text{Unit}))$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{putLive} : C \rrbracket_{\text{val}, \mathcal{R} \zeta}$	$= \lambda \psi \in \zeta. \lambda v \in \mathbb{Z}. \lambda s \in \mathbb{Z}. ((\text{Int}, s + \{\xi \mapsto v\}))$
raise	$: \Pi_{\text{ss}} \xi (\Pi_{\text{sym}} \delta : \xi (\text{Exception}^{\star_S} \xi (\text{singleton } \delta) \alpha))$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{raise} : C \rrbracket_{\text{val}, \mathcal{R} \zeta}$	$= \lambda \psi \in \zeta. \text{inr } \psi$
handle	$: \Pi_{\text{ss}} \xi (\Pi_{\text{sym}} \delta : \xi (\text{Exception}^{\star_S} \xi F \alpha$ $\rightarrow \text{Exception}^{\star_S} \xi G \alpha \rightarrow \text{Exception}^{\star_S} \xi (\text{excHandle } \delta F G) \alpha))$
$\llbracket \mathcal{J} \vdash_{\text{val}} \text{handle} : C \rrbracket_{\text{val}, \mathcal{R} \zeta}$	$= \lambda \psi \in \zeta.$ $\lambda m \in \llbracket \mathcal{J}_{\Sigma, \mathcal{S}, \Delta, \Phi} \vdash_{\text{type}} \text{Exception}^{\star_S} \xi F \alpha \rrbracket_{\text{type}, \mathcal{R}_{\Sigma, \mathcal{S}, \Delta, \Phi}}.$ $\lambda h \in \llbracket \mathcal{J}_{\Sigma, \mathcal{S}, \Delta, \Phi} \vdash_{\text{type}} \text{Exception}^{\star_S} \xi G \alpha \rrbracket_{\text{type}, \mathcal{R}_{\Sigma, \mathcal{S}, \Delta, \Phi}}.$ $\begin{cases} m, & m \neq \text{inr } \psi \\ h, & m = \text{inr } \psi \end{cases}$

Figure D.8. Semantics of the graded monad constants. The type of each constant is stated above its semantics and is abbreviated as C in each of the judgements.

$$\begin{aligned}
& \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{gr}} \text{rdPerm} \triangleleft \text{StPerm}^\circ S \ \& \ \text{Sym} \Rightarrow \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} = \\
& \quad \lambda \psi \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \lambda \psi' \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \begin{cases} \text{ro}, & \psi' = \psi \\ \text{pure}, & \psi' \neq \psi \end{cases} \\
& \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{gr}} \text{wrPerm} \triangleleft \text{StPerm}^\circ S \ \& \ \text{Sym} \Rightarrow \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} = \\
& \quad \lambda \psi \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \lambda \psi' \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \begin{cases} \text{wo}, & \psi' = \psi \\ \text{pure}, & \psi' \neq \psi \end{cases} \\
& \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{gr}} \text{rdLive} \triangleleft \text{StLive}^\circ S \ \& \ \text{Sym} \Rightarrow \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} = \\
& \quad \lambda \psi \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \lambda s \in \mathcal{P}(\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}). \ s \cup \{\psi\} \\
& \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{gr}} \text{wrLive} \triangleleft \text{StLive}^\circ S \ \& \ \text{Sym} \Rightarrow \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} = \\
& \quad \lambda \psi \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \lambda s \in \mathcal{P}(\llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}). \ s \setminus \{\psi\} \\
& \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{gr}} \text{singleton} \triangleleft \text{Exception}^\circ S \ \& \ \text{Sym} \Rightarrow \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} = \\
& \quad \lambda \psi \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \{\psi\} \\
& \llbracket \Sigma \mid \mathcal{S} \mid \Delta \mid \Phi \vdash_{\text{gr}} \text{excHandle} \triangleleft \text{Exception}^\circ S \ \& \ \text{Sym} \Rightarrow \text{Eff} \Rightarrow \text{Eff} \rrbracket_{\text{gr}} \mathcal{R} = \\
& \quad \lambda \psi \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \mathcal{R}_{\Sigma, S}. \ \lambda \phi_1 \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{alg}} \text{Exception}^\circ S \rrbracket^{\text{alg}} \mathcal{R}_{\Sigma, S}. \\
& \quad \lambda \phi_2 \in \llbracket \Sigma \mid \mathcal{S} \vdash_{\text{alg}} \text{Exception}^\circ S \rrbracket^{\text{alg}} \mathcal{R}_{\Sigma, S}. \ (\phi_1 \setminus \{\psi\}) \cup \phi_2
\end{aligned}$$

Figure D.9. Denotations of grade-kinded operators for the three graded monad builders.

Extended examples

This Appendix contains elaborations and further examples related to topics covered in the core part of the thesis.

The two examples presented here are:

- the translation of **LiveState_S** (following on Chapter 3);
- the graded monad transformer corresponding to **HistState_S** (following on Chapter 7).

E.1 Live variables

The translation of **LiveState_S** is shown on Figures E.1 and E.2.

E.2 History-graded state monad transformer

Example E.1. Let $\mathbb{C}$ be a cartesian closed category and let $S \in \mathbb{C}$. The true strong graded monad transformer **HistStateT_S** represents the addition of a new state (whose type is represented by S) and the added analysis is that of the maximum number of writes performed on this state by a computation. Given a strong graded monad $\mathbf{T} = (T, \mu, \eta, t) : (E, \leq, \odot, \mathbf{1}) \rightarrow \mathbf{Endo}(\mathbb{C})$, the strong graded monad $\widehat{\mathbf{T}} = \mathbf{HistStateT}_S \mathbf{T}$ has a grading algebra that is just a direct (cartesian) product of $(\mathbb{N}_\omega, \leq_\omega, +_\omega, 0)$ and $(E, \leq, \odot, \mathbf{1})$. Assuming t^f is the ‘flipped’ version of t , meaning $t_{A,B}^f = t_{B,A}$ for all $A, B \in \mathbb{C}$, the graded monad $\widehat{\mathbf{T}}$ is defined as follows:

$$\begin{aligned}
 \widehat{T}\langle n, e \rangle &= (Te((-) \times S^{\leq_\omega n+1}))^S \quad (\text{for } n \in \mathbb{N}) \\
 \widehat{T}\langle \infty, e \rangle &= (Te((-) \times \mathbf{List}(S)))^S \\
 (\widehat{T}(\langle n_1, e \rangle \leq \langle n_2, e \rangle))_A &= \text{id}_{\widehat{T}\langle n_1, e \rangle A} \\
 (\widehat{T}(\langle n, e_1 \rangle \leq \langle n, e_2 \rangle))_A &= \lambda((T(e_1 \leq e_2))_{S, Te_1(A \times S^{\leq_\omega n+1})} \circ \text{ev}_{S, Te_1(A \times S^{\leq_\omega n+1})}) \\
 (\widehat{\mu}_{\langle n_1, e_1 \rangle, \langle n_2, e_2 \rangle})_X &= \lambda((\mu_{e_1, e_2})_{X \times S^{\leq_\omega 1+\omega n_1+\omega n_2}} \circ Te_1(\text{id}_X \times \text{append}_S^{1+\omega n_2, n_1}) \\
 &\quad \circ Te_1 t_{X \times S^{\leq_\omega 1+\omega n_2}, S^{\leq_\omega n_1}}^f \\
 &\quad \circ Te_1 \langle \text{ev}_{S, Te_2(X \times S^{\leq_\omega 1+\omega n_2})} \circ \langle \text{hd} \circ \pi_2, \pi_1 \rangle, \text{tl} \circ \pi_2 \rangle \\
 &\quad \circ \text{ev}_{S, Te_1((Te_2(X \times S^{\leq_\omega 1+\omega n_2}))^S \times S^{\leq_\omega 1+\omega n_1})}) \\
 \widehat{\eta}_A &= \lambda \eta_{A \times S}
 \end{aligned}$$

$$\begin{aligned}
\langle\!\langle \text{LiveState}_S^{\text{kill}} A \rangle\!\rangle_{\text{type}} &= \langle\!\langle A \rangle\!\rangle_{\text{type}} \times \langle\!\langle S \rangle\!\rangle_{\text{type}} \\
\langle\!\langle \text{LiveState}_S^{\text{id}} A \rangle\!\rangle_{\text{type}} &= \langle\!\langle A \rangle\!\rangle_{\text{type}} \times (\langle\!\langle S \rangle\!\rangle_{\text{type}} + \text{Unit}) \\
\langle\!\langle \text{LiveState}_S^{\text{gen}} A \rangle\!\rangle_{\text{type}} &= \langle\!\langle S \rangle\!\rangle_{\text{type}} \rightarrow (\langle\!\langle A \rangle\!\rangle_{\text{type}} \times \langle\!\langle S \rangle\!\rangle_{\text{type}})
\end{aligned}$$

$$\begin{aligned}
(\text{RET}) \quad \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{return } M : \text{LiveState}_S^{\text{id}} A} &\quad \bullet \quad \text{inl } t_1 : \langle\!\langle A \rangle\!\rangle_{\text{type}} + (\langle\!\langle S \rangle\!\rangle_{\text{type}} + \text{Unit}) \\
(\text{GET}) \quad \frac{}{\Gamma \vdash \text{get} : \text{LiveState}_S^{\text{gen}} S} &\quad \bullet \quad \text{fn } s : \langle\!\langle S \rangle\!\rangle_{\text{type}} \Rightarrow \langle s, s \rangle \\
(\text{PUT}) \quad \frac{}{\Gamma \vdash \text{put} : S \rightarrow \text{LiveState}_S^{\text{kill}} \text{Unit}} &\quad \bullet \quad \text{fn } s : \langle\!\langle S \rangle\!\rangle_{\text{type}} \Rightarrow \langle \langle \rangle, s \rangle \\
(\text{RUN-kill}) \quad \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{kill}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} &\quad \bullet \quad \text{fn } s : \langle\!\langle S \rangle\!\rangle_{\text{type}} \Rightarrow \text{proj}_1 t_1 \\
(\text{RUN-id}) \quad \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{id}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} &\quad \bullet \quad \text{fn } s : \langle\!\langle S \rangle\!\rangle_{\text{type}} \Rightarrow \text{proj}_1 t_1 \\
(\text{RUN-gen}) \quad \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{gen}} A}{\Gamma \vdash \text{run } M : S \rightarrow A} &\quad \bullet \quad \text{fn } s : \langle\!\langle S \rangle\!\rangle_{\text{type}} \Rightarrow \text{proj}_1 (t_1 s) \\
(\text{SUB-REFL}) \quad \frac{\Gamma \vdash M : \text{LiveState}_S^e A}{\Gamma \vdash \text{sub}_{e,e} M : \text{LiveMultiState}_S^e A} &\quad \bullet \quad t \\
(\text{SUB-kill-id}) \quad \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{kill}} A}{\Gamma \vdash \text{sub}_{\text{kill},\text{id}} M : \text{LiveState}_S^{\text{id}} A} &\quad \bullet \\
&\quad \langle \text{proj}_1 t_1, \text{inl } (\text{proj}_2 t_1) : \langle\!\langle S \rangle\!\rangle_{\text{type}} + \text{Unit} \rangle \\
(\text{SUB-kill-gen}) \quad \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{kill}} A}{\Gamma \vdash \text{sub}_{\text{kill},\text{gen}} M : \text{LiveState}_S^{\text{gen}} A} &\quad \bullet \\
&\quad \text{fn } s : \langle\!\langle S \rangle\!\rangle_{\text{type}} \Rightarrow t_1 \\
(\text{SUB-id-gen}) \quad \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{id}} A}{\Gamma \vdash \text{sub}_{\text{id},\text{gen}} M : \text{LiveState}_S^{\text{gen}} A} &\quad \bullet \\
&\quad \text{fn } s : \langle\!\langle S \rangle\!\rangle_{\text{type}} \Rightarrow \langle \text{proj}_1 t_1, \text{case } \text{proj}_2 t \text{ of } (\text{inl } x_1 \Rightarrow x_1 \mid \text{inr } x_2 \Rightarrow s) \rangle
\end{aligned}$$

Figure E.1. The translation of LiveState_S into Target – everything except binding. Fresh names are assumed for all introduced variables.

$$\begin{array}{c}
\text{(BIND-kill-kill)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{kill}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^{\text{kill}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^{\text{kill}} B} \quad \sim \\
t_2\{(\text{proj}_1 t_1)/x\} \\
\\
\text{(BIND-kill-id)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{kill}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^{\text{id}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^{\text{kill}} B} \quad \sim \\
\langle \text{proj}_1 t_2, \text{case } \text{proj}_2 t_2 \text{ of } (\text{inl } x_1 \Rightarrow x_1 \mid \text{inr } x_2 \Rightarrow \text{proj}_2 t_1) \rangle \{(\text{proj}_1 t_1)/x\} \\
\\
\text{(BIND-kill-gen)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{kill}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^{\text{gen}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^{\text{kill}} B} \quad \sim \\
(t_2 (\text{proj}_2 t_1)) \{(\text{proj}_1 t_1)/x\} \\
\\
\text{(BIND-id)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{id}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^e B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^e B} \quad \sim \\
t_2\{(\text{proj}_1 t_1)/x\} \quad (\text{for } e \in \{\text{kill}, \text{id}\}) \\
\\
\text{(BIND-id-gen)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{id}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^{\text{gen}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^{\text{gen}} B} \quad \sim \\
\text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow (\text{case } \text{proj}_2 t_1 \text{ of } (\text{inl } x_1 \Rightarrow t_2 x_1 \mid \text{inr } x_2 \Rightarrow t_2 s)) \{(\text{proj}_1 t_1)/x\} \\
\\
\text{(BIND-gen-kill)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{gen}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^{\text{kill}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^{\text{gen}} B} \quad \sim \\
\text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow t_2 \{(\text{proj}_1 (t_1 s))/x\} \\
\\
\text{(BIND-gen-id)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{kill}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^{\text{id}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^{\text{kill}} B} \quad \sim \\
\text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow \langle \text{proj}_1 t_2, \text{case } \text{proj}_2 t_2 \text{ of } (\text{inl } x_1 \Rightarrow \text{proj}_2 t_2 \mid \text{inr } x_2 \Rightarrow \text{proj}_2 (t_1 s)) \rangle \{(\text{proj}_1 (t_1 s))/x\} \\
\\
\text{(BIND-gen-gen)} \frac{\Gamma \vdash M : \text{LiveState}_S^{\text{gen}} A \quad \Gamma, x : A \vdash N : \text{LiveState}_S^{\text{gen}} B}{\Gamma \vdash \text{bind } x : A \Leftarrow M \text{ with } N : \text{LiveState}_S^{\text{gen}} B} \quad \sim \\
\text{fn } s : \langle\langle S \rangle\rangle_{\text{type}} \Rightarrow (t_2 (\text{proj}_2 (t_1 s))) \{(\text{proj}_1 (t_1 s))/x\}
\end{array}$$

Figure E.2. The translation of binding for LiveState_S .

The multiplication natural transformation $\widehat{\mu}$, the most complicated part of the definition, is such that $(\widehat{\mu}_{\langle n_1, e_1 \rangle, \langle n_2, e_2 \rangle})_X$ is the unique morphism such that the diagram on Figure E.3 commutes.

$$\begin{array}{ccc}
Te_1((Te_2(X \times S^{\leq_{\omega} 1 +_{\omega} n_2}))^S \times S^{\leq_{\omega} 1 +_{\omega} n_1})^S \times S & \xrightarrow{(\widehat{\mu}_{\langle n_1, e_1 \rangle, \langle n_2, e_2 \rangle})^X \times \text{id}_S} & (T(e_1 \odot e_2)(X \times S^{\leq_{\omega} 1 +_{\omega} n_1 +_{\omega} n_2}))^S \times S \\
\downarrow \text{ev}_{S, Te_1((Te_2(X \times S^{\leq_{\omega} 1 +_{\omega} n_2}))^S \times S^{\leq_{\omega} 1 +_{\omega} n_1})} & & \downarrow \text{ev}_{S, T(e_1 \odot e_2)(X \times S^{\leq_{\omega} 1 +_{\omega} n_1 +_{\omega} n_2})} \\
Te_1((Te_2(X \times S^{\leq_{\omega} 1 +_{\omega} n_2}))^S \times S^{\leq_{\omega} 1 +_{\omega} n_1}) & & \\
\downarrow Te_1(\text{ev}_{S, Te_2(X \times S^{\leq_{\omega} 1 +_{\omega} n_2})} \circ (\text{hd} \circ \pi_2^{S, n_1})) & & \\
Te_1(Te_2(X \times S^{\leq_{\omega} 1 +_{\omega} n_2}) \times S^{\leq_{\omega} n_1}) & & \\
\downarrow Te_1((\text{id}_X \times \text{append}_S^{1+n_2, n_1}) \circ f_{X \times S^{\leq_{\omega} 1 +_{\omega} n_2}, \leq_{\omega} n_1}) & & \\
Te_1(Te_2(X \times S^{\leq_{\omega} 1 +_{\omega} n_2} \times S^{\leq_{\omega} n_1})) & \xrightarrow{(\mu_{e_1, e_2})_{X \times S^{\leq_{\omega} 1 +_{\omega} n_1 +_{\omega} n_2}}} & T(e_1 \odot e_2)(X \times S^{\leq_{\omega} 1 +_{\omega} n_1 +_{\omega} n_2})
\end{array}$$

Figure E.3. Definition of $\widehat{\mu}$ (Example E.1).

Additional proofs

F.1 Proof of soundness of the System G equational theory

Proving Theorem 6.1 is done using a simple induction on derivations of $\equiv$, but the proof relies on some certain other properties. The β -equivalence cases in the proof of the soundness theorem are proved by using the following *soundness of substitution* result:

Theorem F.1. The following denotational equalities hold:

- (1) $\llbracket \mathcal{J} \vdash_{\text{val}} M\{N/x\} : A \rrbracket_{\text{val}} \mathcal{R} = \llbracket \mathcal{J} \{ \Gamma \mapsto x:B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} (\mathcal{R} \{ \Gamma \mapsto \mathcal{R}_\Gamma[x \mapsto \llbracket \mathcal{J} \vdash_{\text{val}} N : B \rrbracket_{\text{val}} \mathcal{R}] \});$
- (2) $\llbracket \mathcal{J} \vdash_{\text{val}} M\{F/r\} : A \rrbracket_{\text{val}} \mathcal{R} = \llbracket \mathcal{J} \{ \Phi \mapsto r:[T]S, \mathcal{J}_\Phi \} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} (\mathcal{R} \{ \Phi \mapsto \mathcal{R}_\Phi[r \mapsto \llbracket \mathcal{J}_{\Sigma, S, \Delta, \Phi} \vdash_{\text{gr}} F \triangleleft [T]S \& \text{Eff} \rrbracket_{\text{gr}} \});$
- (3) $\llbracket \mathcal{J} \vdash_{\text{val}} M\{\Psi/\delta\} : A \rrbracket_{\text{val}} \mathcal{R} = \llbracket \mathcal{J} \{ \Delta \mapsto \delta:S, \mathcal{J}_\Delta \} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} (\mathcal{R} \{ \Delta \mapsto \mathcal{R}_\Delta[\delta \mapsto \llbracket \mathcal{J}_{\Sigma, S, \Delta} \vdash_{\text{sym}} \Psi \in S \rrbracket_{\text{sym}} \});$
- (4) $\llbracket \mathcal{J} \vdash_{\text{val}} M\{S/\xi\} : A \rrbracket_{\text{val}} \mathcal{R} = \llbracket \mathcal{J} \{ \mathcal{S} \mapsto \xi, \mathcal{J}_\mathcal{S} \} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} (\mathcal{R} \{ \mathcal{S} \mapsto \mathcal{R}_\mathcal{S}[\xi \mapsto \llbracket \mathcal{J}_{\Sigma, \mathcal{S}} \vdash_{\text{ss}} S \rrbracket_{\text{ss}} \});$

Proof. By induction on typing judgements. The inductive property for case (1) is:

$$\begin{aligned} \Xi(\mathcal{J} \{ \Gamma \mapsto x:B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M : A) &\equiv \forall N. \mathcal{J} \vdash_{\text{val}} N : B \\ &\Rightarrow \llbracket \mathcal{J} \vdash_{\text{val}} M\{N/x\} : A \rrbracket_{\text{val}} \mathcal{R} \\ &= \llbracket \mathcal{J} \{ \Gamma \mapsto x:B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} \\ &\quad (\mathcal{R} \{ \Gamma \mapsto \mathcal{R}_\Gamma[x \mapsto \llbracket \mathcal{J} \vdash_{\text{val}} N : B \rrbracket_{\text{val}} \mathcal{R}] \}) \end{aligned}$$

We give one case in part (1) of the statement as an example:

(app) Assume $\Xi(\mathcal{J} \{ \Gamma \mapsto x:B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M_1 : A_1 \rightarrow A_2)$ and $\Xi(\mathcal{J} \{ \Gamma \mapsto x:B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M_2 : A_1)$. Let N be a term satisfying $\mathcal{J} \vdash_{\text{val}} N : B$. Since $(M_1 M_2)\{N/x\}$ is the same as $M_1\{N/x\} M_2\{N/x\}$, we get:

$$\begin{aligned} \llbracket \mathcal{J} \vdash_{\text{val}} (M_1 M_2)\{N/x\} : A_2 \rrbracket_{\text{val}} \mathcal{R} &= (\llbracket \mathcal{J} \vdash_{\text{val}} M_1\{N/x\} : A_1 \rightarrow A_2 \rrbracket_{\text{val}} \mathcal{R}) \\ &\quad (\llbracket \mathcal{J} \vdash_{\text{val}} M_2\{N/x\} : A_1 \rrbracket_{\text{val}} \mathcal{R}) \\ &= ((\llbracket \mathcal{J} \{ \Gamma \mapsto x : B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M_1 : A_1 \rightarrow A_2 \rrbracket_{\text{val}} \mathcal{R}) \\ &\quad \mathcal{R} \{ \Gamma \mapsto \mathcal{R}_\Gamma[x \mapsto \llbracket \mathcal{J} \vdash_{\text{val}} N : B \rrbracket_{\text{val}} \mathcal{R}] \}) \\ &\quad ((\llbracket \mathcal{J} \{ \Gamma \mapsto x : B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M_2 : A_1 \rrbracket_{\text{val}} \mathcal{R}) \\ &\quad \mathcal{R} \{ \Gamma \mapsto \mathcal{R}_\Gamma[x \mapsto \llbracket \mathcal{J} \vdash_{\text{val}} N : B \rrbracket_{\text{val}} \mathcal{R}] \}) \\ &= (\llbracket \mathcal{J} \{ \Gamma \mapsto x : B, \mathcal{J}_\Gamma \} \vdash_{\text{val}} M_1 M_2 : A_2 \rrbracket_{\text{val}} \mathcal{R}) \\ &\quad \mathcal{R} \{ \Gamma \mapsto \mathcal{R}_\Gamma[x \mapsto \llbracket \mathcal{J} \vdash_{\text{val}} N : B \rrbracket_{\text{val}} \mathcal{R}] \} \end{aligned}$$

as required.

Other cases proceed similarly. □

Proof of Theorem 6.1. By induction on derivations for judgements $\mathcal{J} \vdash M \equiv M' : A$, with property:

$$\Xi(\mathcal{J} \vdash M \equiv M' : A) : \llbracket \mathcal{J} \vdash_{\text{val}} M : A \rrbracket_{\text{val}} = \llbracket \mathcal{J} \vdash_{\text{val}} M' : A \rrbracket_{\text{val}}$$

□

F.2 Proof of correctness of type inference for ML_G

An important tool in the proof that the type inference algorithm presented in Section 6.3.2 and Figure 6.7 is correct is the following lemma:

Lemma F.1 (Substitution). If $\Sigma \mid \Delta \mid \Gamma \Vdash M : A$ is derivable, then so is $\Sigma \mid \Delta \mid \theta\Gamma \Vdash e : \theta A$ for every type and grade substitution θ .

Proof. Consider the property $\Phi(\Sigma, \Delta, \Gamma, M, A) \equiv \forall \theta. \Sigma \mid \Delta \mid \theta\Gamma \Vdash M : \theta(A)$. We want to show that it holds for all $\Sigma, \Delta, \Gamma, M, A$ satisfying $\Sigma \mid \Delta \mid \Gamma \Vdash M : A$. We prove this by rule induction on the definition of typing judgements.

(var) Suppose $\Sigma \mid \Delta \mid \Gamma \Vdash x : A$ holds due to $\Gamma(x) = \sigma$ and $\sigma > \tau$. In other words, σ is of the form $(\forall_{\text{eff}} R)(\forall_{\text{type}} V)A'$, with $A = \theta'A'$ for some substitution $\mathcal{S}'$ (whose only non-fixed points must be elements of R and V). For any substitution θ , there is a substitution $\tilde{\theta}$ that is the same as θ but does leaves all type and effect variables in V and R alone – in other words $\theta\Gamma(x) = (\forall_{\text{eff}} R')(\forall_{\text{type}} V')(\tilde{\theta}A')$. Here the sets R' and V' are introduced to rename the $\forall$ -bound variables using fresh names and prevent ambiguity (following Damas and Milner). We have $(\forall_{\text{eff}} R)(\forall_{\text{type}} V)(\theta A') > \theta A$ because $\theta A = \theta''\tilde{\theta}A'$ for the following θ'' :

$$\theta''(r) = \begin{cases} \theta\theta'(r), & r \in R' \\ r & r \notin R' \end{cases} \quad \theta''(\alpha) = \begin{cases} \theta\theta'(\alpha), & \alpha \in A' \\ \alpha & \alpha \notin A' \end{cases}$$

To show this works, first use the fact that $A = \theta'A'$, and then consider how $\theta\theta'$ and $\theta''\tilde{\theta}$ act on type variables and effect variables. In the case of a bound type variable α , $\theta''(\tilde{\theta}(\alpha)) = \theta''(\alpha) = \theta\theta'(\alpha)$. When α is free, $\theta'(\alpha) = \alpha$ and θ must map α to a type that contains no type variables in A' , so all type variables appearing in $\theta(\alpha)$ are left fixed by θ'' – hence $\theta''(\theta(\alpha)) = \theta(\alpha)$. The argument for effect variables is analogous. Hence we are done.

(sub) Assume $\Phi(\Sigma, \Delta, \Gamma, M, A), \Sigma \mid \Delta \mid \Gamma \Vdash M : A$ and $A <: B$. Then $\Sigma \mid \Delta \mid \Gamma \Vdash M : B$ by (sub). Consider any substitution θ . By the induction hypothesis, we have $\Sigma \mid \Delta \mid \theta\Gamma \Vdash M : \theta A$. To proceed, we use the following result:

Lemma F.2. For any substitution θ and types A and B , $\Sigma \mid \Delta \Vdash A <: B$ implies $\Sigma \mid \Delta \Vdash \theta A <: \theta B$.

Proof. Easy rule induction on the definition of $<:$. The only case that requires special attention is (GRADED- $<:$) because we want to show that $\Sigma \mid \Delta \Vdash_{\text{sub}} F \sqsubseteq F'$ implies $\Sigma \mid \Delta \Vdash_{\text{sub}} \theta F \sqsubseteq \theta F'$. If F and F' contain any effect variables, then the conclusion must hold for all possible valuations of these effect variables, including the ones that correspond to the application of substitution θ . □

Hence $\Sigma \mid \Delta \Vdash \theta A <: \theta B$. Finally, $\Sigma \mid \Delta \mid \theta \Gamma \Vdash M : \theta B$.

Other cases. Proceed similarly.

□

Proof of Theorem 6.2. We prove the two parts of this theorem by induction on **MLG** terms M .

1. Consider the following property of expressions:

$$\Phi(M) \equiv \forall \Sigma, \Delta, \Gamma, \theta, A. \quad \begin{array}{l} \text{if } \text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M) = (\theta, A), \\ \text{then } \Sigma \mid \Delta \mid \theta \Gamma \Vdash M : A \text{ is derivable} \end{array}$$

We prove this by induction on the structure of terms M .

(new) Assume $\Phi(M)$ and consider the expression **new** X **within** M . Let $\Sigma, \Delta, \Gamma, \theta, A$ such that $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M) = (\theta, A)$. By definition of $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M)$, the type A is the same as the type τ inferred for M in context $X, \Sigma \mid \Delta \mid \text{prune}(\Gamma)$. By instantiating the inductive hypothesis $\Phi(M)$ with X, Σ, Δ and $\text{prune}(\Gamma)$, we infer that $X, \Sigma \mid \Delta \mid \theta(\text{prune}(\Gamma)) \Vdash M : \tau$. By the **(NEW)** typing rule we infer $\Sigma \mid \Delta \mid \theta \Gamma \Vdash M : \tau$, and since τ and A are the same, we conclude $\Phi(\text{new } X \text{ within } M)$.

(sym-app) Assume $\Phi(M)$ and consider the expression $M \Psi$. Let $\Sigma, \Delta, \Gamma, \theta, A$ such that $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M \Psi) = (\theta, A)$. By definition of $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M \Psi)$, M has type $\Pi_{\text{sym}} \delta (A')$ for some δ in variable context $\theta \Gamma$ and A is the same as $A'\{\Psi/\delta\}$. The side condition also specifies that Ψ is a valid symbol given the symbol context $\Sigma \mid \Delta$. By instantiating the inductive hypothesis $\Phi(M)$ with Σ, Δ, Γ and $\Pi_{\text{sym}} \delta (A')$ we infer that $\Sigma \mid \Delta \mid \theta \Gamma \Vdash M : \Pi_{\text{sym}} \delta (A')$. By the **(SYM-APP)** typing rule we infer $\Sigma \mid \Delta \mid \theta \Gamma \Vdash M : A'\{\Psi/\delta\}$, and since $A'\{\Psi/\delta\}$ is just A , we conclude $\Phi(M \Psi)$.

(var) Assume $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash x) = (\theta, A)$. By definition of $\text{infer}(\cdot)$, we have $\theta = \text{Id}$ and $\Gamma(x) = (\forall_{\text{eff}} R)(\forall_{\text{type}} V)A$ for some R and A . Since $\Gamma(x) > A$, by **(VAR)** we reach the desired conclusion.

(fn) Assume $\Phi(M)$ and let $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash \text{fn } x \Rightarrow M) = (\theta, A)$. By definition of $\text{infer}(\cdot)$, there is a fresh type variable α that does not appear in Σ, Δ and Γ such that $(\theta, A') = \text{infer}(\Sigma \mid \Delta \mid x : \alpha, \Gamma \Vdash e)$ and $A = \theta(\alpha) \rightarrow A'$. By the inductive hypothesis, $\Sigma \mid \Delta \mid x : \theta(\alpha), \theta \Gamma \Vdash M : A$. Thus, by the **(FN)** rule, we conclude $\Sigma \mid \Delta \mid \theta \Gamma \Vdash M : \theta(\alpha) \rightarrow A$.

(app) Assume $\Phi(M_1)$ and $\Phi(M_2)$. Let $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M_1 M_2) = (\theta, A)$. Following the definition of $\text{infer}(\cdot)$, we have:

- $(\theta_1, A_1) = \text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M_1)$;
- $(\theta_2, A_2) = \text{infer}(\Sigma \mid \Delta \mid \theta_1 \Gamma \Vdash M_2)$;
- $\theta_3 = \text{unify}(\theta_2 A_1, A_2 \rightarrow \alpha)$ for a fresh type variable α ;
- A is $\theta_3(\alpha)$ and $\theta = \theta_3 \theta_2 \theta_1$.

From the induction hypothesis $\Phi(M_1)$ we conclude $\Sigma \mid \Delta \mid \theta_1 \Gamma \Vdash M_1 : A_1$. By the Substitution Lemma, we conclude $\Sigma \mid \Delta \mid \theta_3 \theta_2 \theta_1 \Gamma \Vdash M_1 : \theta_3 \theta_2 A_1$. By definition of unification, $\theta_3 \theta_2 A_1$ is the same as $\theta_3 A_2 \rightarrow \theta_3(\alpha)$, and thus $\Sigma \mid \Delta \mid \theta_3 \theta_2 \theta_1 \Gamma \Vdash M_1 : \theta_3 A_2 \rightarrow \theta_3(\alpha)$.

From $\Phi(M_2)$ we conclude $\Sigma \mid \Delta \mid \theta_2\theta_1\Gamma \Vdash M_1 : A_2$, and applying the Substitution Lemma leads to the conclusion $\Sigma \mid \Delta \mid \theta_3\theta_2\theta_1\Gamma \Vdash M_1 : \theta_3A_2$. Hence, by (APP), we conclude $\Sigma \mid \Delta \mid \theta_3\theta_2\theta_1\Gamma \Vdash M_1 M_2 : \theta_3(\alpha)$.

(ret) Assume $\Phi(M)$ and let $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash \text{return } M) = (\theta, A)$. By definition of $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash \text{return } M)$, the type A is necessarily of the form T^1A' , and furthermore $(\theta, A') = \text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M)$. By $\Phi(M)$, we have $\Sigma \mid \Delta \mid \theta\Gamma \Vdash M : A'$, and by (RET) we conclude $\Sigma \mid \Delta \mid \theta\Gamma \Vdash \text{return } M : T^1A'$.

(bind) Assume $\Phi(M_1)$ and $\Phi(M_2)$ and consider the expression $\text{bind } x \Leftarrow M_1 \text{ with } M_2$. Let $\Sigma, \Delta, \Gamma, \theta, A$ such that $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash \text{bind } x \Leftarrow M_1 \text{ with } M_2) = (\theta, A)$. By definition of $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash \text{bind } x \Leftarrow M_1 \text{ with } M_2)$ there exist type variables α, β and effect variables r, s , and substitutions $\theta_1, \theta_2, \theta_3, \theta_4$ such that $\theta = \theta_4\theta_3\theta_2\theta_1$. Furthermore, by the inductive hypothesis, $\Sigma \mid \Delta \mid \theta_1\Gamma \Vdash M_1 : A_1$ and $\Sigma \mid \Delta \mid \theta_2(x : A_1, \theta_1\Gamma) \Vdash M_2 : A_2$.

By the Substitution Lemma, $\Sigma \mid \Delta \mid \theta_3\theta_2\theta_1\Gamma \Vdash M_1 : \theta_3\theta_2A_1$, or in other words, $\Sigma \mid \Delta \mid \theta_3\theta_2\theta_1\Gamma \Vdash M_1 : T^{\theta_3(r)}(\theta_3(\alpha))$. Applying the Substitution Lemma again we get $\Sigma \mid \Delta \mid \theta\Gamma \Vdash M_1 : T^{\theta_4\theta_3(r)}(\theta_4\theta_3(\alpha))$.

Similarly we conclude $\Sigma \mid \Delta \mid \theta\Gamma \Vdash M_2 : T^{\theta_4(s)}(\theta_4(\beta))$.

By (BIND), we conclude $\Sigma \mid \Delta \mid \theta\Gamma \Vdash M_2 : T^{\theta_4\theta_3(r) \odot \theta_4(s)}(\theta_4(\beta))$. By simplifying $\theta_4(\beta)$ we get the desired conclusion.

Other cases are similar.

2. Consider the following property:

$$\begin{aligned} \Xi(\Sigma, \Delta, \Gamma, M, A') &\equiv \forall \theta, A. \text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash M) = (\theta, A) \\ &\Rightarrow \exists A''. (\Sigma \mid \Gamma \Vdash A'' <: A' \wedge \sigma > A'') \end{aligned}$$

where $\sigma = (\forall_{\text{eff}} R)(\forall_{\text{type}} V)A$, with $R = \text{fev}(A) \setminus \text{fev}(\Gamma)$ and $V = \text{ftv}(A) \setminus \text{ftv}(\Gamma)$. We show that this property holds for all $\Sigma, \Delta, \Gamma, M, A'$ satisfying $\Sigma \mid \Delta \mid \Gamma \Vdash M : A'$ by induction on rules that define the typing relation.

(sym-fn) Assume $\Xi(\Sigma, (\delta, \Delta), \Gamma, M, A')$ and furthermore that $\Sigma \mid \delta, \Delta \mid \Gamma \Vdash M : A'$ is derivable. By (SYM-FN), $\Sigma \mid \Delta \mid \Gamma \Vdash \text{sym } \delta \Rightarrow M : \Pi_{\text{sym}} \delta (A')$ is also derivable. Assume $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash \text{sym } \delta \Rightarrow M) = (\theta, A)$. By definition of $\text{infer}(\cdot)$, this means that A is $\Pi_{\text{sym}} \delta (\tilde{A})$ and that $\text{infer}(\Sigma \mid \delta, \Delta \mid \Gamma \Vdash M) = (\theta, \tilde{A})$ for some type $\tilde{A}$. By the inductive hypothesis, there exists some type A'' such that $\Sigma \mid \Delta \Vdash A'' <: A'$ and $\tilde{\sigma} > A''$, where $\tilde{\sigma} = (\forall_{\text{eff}} R)(\forall_{\text{type}} V)\tilde{A}$ (whereas $\sigma = (\forall_{\text{eff}} R)(\forall_{\text{type}} V)A$).

Now consider the type $\Pi_{\text{sym}} \delta (A'')$. By rule (DEP-<:) we have $\Sigma \mid \Delta \Vdash \Pi_{\text{sym}} \delta (A'') <: \Pi_{\text{sym}} \delta (A')$. Furthermore, since $\tilde{\sigma} > A''$, there exists a substitution θ' such that $\theta'\tilde{A} = A''$. Then $\theta'(\Pi_{\text{sym}} \delta (\tilde{A})) = \Pi_{\text{sym}} \delta (A'')$, hence $\sigma > \Pi_{\text{sym}} \delta (A'')$. Thus, $\Xi(\Sigma, \Delta, \Gamma, \text{sym } \delta \Rightarrow M, \Pi_{\text{sym}} \delta (A'))$.

(var) Let $(x : \sigma) \in \Gamma$ and $\sigma > A'$. By (VAR), this means $\Sigma \mid \Delta \mid \Gamma \Vdash x : A'$. Now assume $\text{infer}(\Sigma \mid \Delta \mid \Gamma \Vdash x) = (\theta, A)$. By definition of $\text{infer}(\cdot)$, we have that $\sigma = (\forall_{\text{eff}} R)(\forall_{\text{type}} V)A$ for some effect and type variables sets R and V , as well as $\theta = \text{Id}$. Since $<:$ is reflexive, we have $\Sigma \mid \Delta \Vdash A' <: A'$, and we assumed $\sigma > A'$, we conclude $\Xi(\Sigma, \Delta, \Gamma, x, A')$.

Other cases proceed similarly. $\square$