Lifting of Operations in Modular Monadic Semantics

Mauro Jaskelioff

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

August 2009

Abstract

Monads have become a fundamental tool for structuring denotational semantics and programs by abstracting a wide variety of computational features such as side-effects, input/output, exceptions, continuations and non-determinism. In this setting, the notion of a monad is equipped with operations that allow programmers to manipulate these computational effects. For example, a monad for side-effects is equipped with operations for setting and reading the state, and a monad for exceptions is equipped with operations for throwing and handling exceptions.

When several effects are involved, one can employ the incremental approach to modular monadic semantics, which uses monad transformers to build up the desired monad one effect at a time. However, a limitation of this approach is that the effect-manipulating operations need to be manually lifted to the resulting monad, and consequently, the lifted operations are non-uniform. Moreover, the number of liftings needed in a system grows as the product of the number of monad transformers and operations involved.

This dissertation proposes a theory of uniform lifting of operations that extends the incremental approach to modular monadic semantics with a principled technique for lifting operations. Moreover the theory is generalized from monads to monoids in a monoidal category, making it possible to apply it to structures other than monads.

The extended theory is taken to practice with the implementation of a new extensible monad transformer library in Haskell, and with the use of modular monadic semantics to obtain modular operational semantics.

No hay ejercicio intelectual que no sea finalmente inútil. Una doctrina es al principio una descripción verosímil del universo; giran los años y es un mero capítulo—cuando no un párrafo o un nombre—de la historia de la filosofía.

There is no exercise of the intellect which is not, in the final analysis, useless. A philosophical doctrine begins as a plausible description of the universe; with the passage of the years it becomes a mere chapter—if not a paragraph or a name—in the history of philosophy.

Jorge Luis Borges

Acknowledgements

Many people supported me and helped me in writing this thesis and I am deeply thankful to them. Without them, this thesis would not have been possible. I would like to mention:

My supervisors: Neil Ghani, who guided me during my first years as a Ph.D. student and inculcated me with a passion for category theory and with the inescapable moral duty of searching for structure; and Graham Hutton, who excelled at the arduous task of managing my anxieties during the difficult last year.

The School of Computer Science of the University of Nottingham, for giving me financial support.

Tarmo Uustalu and Thorsten Altenkirch, for agreeing to examine my thesis.

Eugenio Moggi, for his inspiring work and for his invaluable help and dedication towards making my ideas precise.

My office mates made my office hours very enjoyable: James Chapman, Peter Hancock, Iain Lain, Conor McBride, Peter Morris, Nicolas Oury, Rawle Prince, and Wouter Swiestra; Ondřej Rypáček, with whom I shared many interesting discussions and fervent categorical pursuits; and all the people at the Functional Programming Lab, who always were willing to discuss research ideas (typically on Friday after 4PM).

Guido Macchi, who inspired me to study computer science in the first place.

My parents Carlos and Graciela, and Raúl and Betty, who travelled great distances to express their support personally. Silvana, Mariela, Javier, Tom, Lara, and Matías, whose confidence in me was always reassuring.

Finally, Emilse, who stood by my side all the time and gave up everything so that I could fulfill my dreams.

Contents

Chapter 1

Introduction

Formal semantics of programming languages are essential for understanding and reasoning about how programs behave. A formal semantics not only provides the means for the analysis and verification of programs, but also it affects language designs decisions as it reveals subtleties arising, for example, from the interaction of different features. Moreover, a clear formal semantics helps the implementor of programming language processors such as compilers and interpreters, by determining the correctness of the implementation and the optimizations performed, and by providing high-level concepts with which to understand and structure the implementation. In a sense, all programmers write programming language processors. Reynolds (1998) observed "any system process that accepts information from human users is such a processor", an idea that it is even more evident today with the rising popularity of domain specific languages (Gibbons 2009).

Modern programming languages combine a great variety of advanced features such as side-effects, input/output, exceptions, continuations, non-determinism and concurrency. Consequently, the formal semantics of languages incorporating these *computational effects* can be very intricate and hence difficult to construct and understand, in effect, hindering the *raison d'être* of a formal semantics.

In this chapter, the monadic approach to semantics is briefly reviewed and the need for libraries for structuring monadic programs is discussed. Then, a description of the structure of the thesis and of its contributions is given.

1.1 The Monadic Approach to Semantics

In order to tackle a complex construction it is beneficial to identify common structure, as common structure gives rise to high-level abstractions which help to make the problem more manageable. Moggi (1989*b*, 1991) observed that monads, a concept arising in category theory, provide an effective abstraction for a great variety of computational effects. It is unsurprising that category theory gave an appropriate abstraction; Reynolds (1980) noted

many years ago that category theory was the appropriate tool for "the central problem" of programming languages, that is "to organize a variety of concepts in a way which exhibits uniformity and generality".

Each monad modelling a computational effect is associated with a set of effect manipulating operations. For example, an exception monad may come with operations for throwing an exception and for handling it, and a state monad may come with operations for reading and updating the state. Then, it is possible to define the semantics of a programming (meta)language for an abstract monad which supports a given set of operations, and only provide a concrete monad at a later stage.

The *monadic approach* to the denotational semantics of a programming language, which has been adapted also to other forms of programming language semantics based on interpreters (Liang, Hudak, and Jones 1995; Wadler 1992) or compilers (Liang and Hudak 1996), consists of three steps (Benton, Hughes, and Moggi 2000; Moggi 1997):

- identify a metalanguage with *computational types*, to hide the interpretation of computational types and operations manipulating *computations*;
- define a translation of the programming language into the metalanguage;
- give a denotational semantics of the metalanguage, by interpreting computational types and operations on computations using a monad and a set of operations associated to it.

However, there is a caveat: when the programming language involves a mixture of computational effects, the number of operations for manipulating computations grows, the monad needed to interpret computational types gets more complex, and the semantics of operations associated to the monad gets more complex, too. To tackle these issues one can adopt a *modular approach*, by providing a set of basic building blocks and operations to build more complex blocks, so that complex monads can be built from simpler ones. Roughly speaking, one can identify two modular approaches:

- the *incremental approach* (taken in Benton, Hughes, and Moggi 2000; Liang, Hudak, and Jones 1995; Moggi 1997) uses unary operations, called monad transformers, which build complex monads by adding one computational feature to a pre-existing monad;
- the *compositional approach* (taken in Hyland, Plotkin, and Power 2006; Lüth and Ghani 2002) uses binary operations, which build complex monads by combining two pre-existing monads.

Both approaches fall short in dealing with operations associated to monads.

The need to *lift* the operations associated to a monad to a combined monad was first identified by Moggi (1989*a*). Liang, Hudak, and Jones (1995) proposed a workaround, namely to

lift in an ad-hoc manner the operations associated to a monad through a monad transformer. However, the workaround is non-modular: the number of liftings of operations grows like the product of the number of monad transformers and operations involved. Alternatively, one may achieve modularity by restricting the format of operations, for instance *algebraic* operations in the sense of Plotkin and Power (2001*b*) are straightforward to lift. However, the applicability of the monadic approach becomes limited if all the operations on computations are required to be algebraic.

The compositional approach fits well with the algebraic view of computational effects advocated in (Hyland, Plotkin, and Power 2006; Plotkin and Power 2001*b*), where monads are replaced by *algebraic theories* (Manes 1976), and combining computational effects is reduced to composition of algebraic theories. Unfortunately, some computational monads are not induced by algebraic theories, most notably the continuation monad; and some operations on computations, such as handling of exceptions, are not algebraic.

The incremental approach, on the other hand, is the most popular among functional programmers, because monad transformers are easy to understand and implement. However, there has been limited progress in addressing the lifting problem. This is precisely the focus of the first part of this thesis, which proposes a theoretical foundation for uniform lifting of operations which is applicable to a wide class of operations and monad transformers.

1.2 Modular Monadic Semantics in Haskell

The power of functional languages lies, to a great extent, in their ability to name and reuse programming idioms (Hughes 1995). This power is often realised in the form of combinator libraries, which consist of a collection of idioms commonly found in the library's application area. Programmers can reuse these idioms and combine them to obtain programs which "very often [...] are correct first time, since they are built by assembling correct components" (Hughes 1995).

Computational effects such as state, exceptions and continuations are usually associated with imperative languages but, with the help of monads, they can be elegantly incorporated into a functional language (Peyton Jones and Wadler 1993). However, obtaining a monad which combines effects can be difficult. Moreover, since monads must satisfy certain coherence conditions, the programmer is faced with the task of verifying these conditions. Obtaining combined effects can be made much easier with a good combinator library for monads.

There are several important qualities that any piece of software should have (Ghezzi, Jazayeri, and Mandrioli 2002). First and foremost is the *correctness* of the implementation, for which a solid-theoretical foundation is a great aid. Other significant qualities are *efficiency*, *portability*, and *usability*, which in combinator libraries is greatly influenced by the

ability of the library to let the programmer abstract from low-level implementation details and think in terms of high-level idioms. Two factors are essential in order to achieve this: the *expressivity* of the library, in the sense that the exposed interface is enough to obtain the desired combinations, without the need to understand the internals of the library; and also the *predictability* of the semantics of the combinators, in the sense that they should behave uniformly and without corner cases. Finally, a library should be *extensible* to cope with the additional requirements that new applications might bring.

Combinator libraries for monads can be built from modular components using monad transformers (Liang, Hudak, and Jones 1995). Current monad transformer libraries, such as mtl (*Monad Transformer Library*), have been very successful in providing useful combinators for constructing monads. However, they have a number of shortcomings. Because no general solution for the problem of lifting operations is known, liftings are done on a case-by-case basis, there is no guarantee that the liftings are uniform, and extending the library is cumbersome. Moreover, the lifting overloading mechanism produces shadowing of operations, and relies essentially on non-portable features. For all these reasons, the predictability, extensibility, expressivity and portability of the library is affected.

The second part of the dissertation introduces Monatron, a transformer library that addresses the issues discussed above. Its implementation builds on the strengths of existing monad transformer libraries and incorporates uniform liftings of operations. Uniform lifting of operations have also been implemented by the author in the library mmtl (*Modular Monad Transformer Library*), but the implementation of mmtl closely follows the design of the mtl and, as a consequence, still suffers from some of its same problems. The desire to eliminate these problems motivated the design of the Monatron library.

Also in the second part, the Monatron library is used to obtain modular interpreters in the style of Liang, Hudak, and Jones (1995) and it is shown that monads can not only be used to structure denotational semantics, but also to structure operational semantics.

1.3 Synopsis

The thesis is divided into two main parts and a conclusion. The first part, which develops a theory of lifting of operations, consists of the following chapters:

Chapter 2 introduces categorical concepts which are needed for the formulation of the theory such as monoidal categories, monoidal functors, and monoidal natural transformations, exponentials and monoids in a monoidal category. Several examples are given.

Chapter 3 presents a formal definition of different classes of operations: H-operations, first-order operations and algebraic operations, and a precise definition of the notion of lifting. It is shown that every algebraic operation can be lifted along a monoid mor-

phism. Additionally, several examples of monads that model computational effects are given along with their associated effect-manipulating operations.

Chapter 4 develops a theory of uniform liftings of operations for monoids in a monoidal category. The theory introduces a hierarchy of monoid transformers, and several lifting theorems are proved for transformers in that hierarchy. Additionally, it is shown that when more than one lifting result is applicable, the liftings coincide.

In the second part of the thesis, the theory is implemented in Haskell and two applications are presented.

Chapter 5 introduces the Haskell library Monatron that implements the theory of uniform liftings, showing how the concepts can be carried out in an actual programming language, and therefore providing evidence that the proposed theory can have a direct impact on programmers.

Chapter 6 reviews how modular interpreters are implemented using monad transformers, providing at the same time an extended example of the use of the Monatron library. The obtained modular semantics also serves as contrast to the modular semantics of the following chapter.

Chapter 7 develops modular operational semantics. It presents an implementation of Turi's functorial operational semantics (Turi 1996; Turi and Plotkin 1997). In this style of operational semantics, the rules that define structural operational semantics are parameterised by a signature functor that describes the syntax and a behaviour functor that describes the observable behaviour. The chapter also presents a notion of modular operational semantics which is obtained by structuring the behaviour functor with a monad and a step functor, effectively applying the techniques from monadic semantics to the case of operational semantics.

As there is no formal semantics of Haskell, it is difficult to establish rigourously the precise correspondence between the theory and the implementation. However, all the proofs in the theory should hold in the implementation under the following assumptions:

- We work with the subset of Haskell without selective strictness constructs such as the `seq` construct. In the presence of `seq` (see, for example, Johann and Voigtländer 2004) η-equivalence does not hold. A rather paradoxical consequence of the lack of extensionality is that in the full version of this language named after the logician Haskell Curry, currying is not an isomorphism.

- There exists a relationally parametric model of Haskell. This assumption will allow us to benefit from free theorems (Wadler 1989) and, for example, obtain that all terms of

type $\forall a.\, \mathsf{F}a \to \mathsf{G}a$ must be natural transformations from the functor F to the functor G. Although no such model is currently known, parametric models of subsets of Haskell have been constructed (Johann and Voigtländer 2009).

Haskell provides more structure than the implementation requires. For example, partial functions are not needed and working with the total fragment of Haskell would be enough. However, the author feels Haskell is a reasonable choice for implementation language as it provides a convenient notation for writing monadic computations, and it is the language on which a monad transformer library has the potential of making the greatest impact.

The final part consists of the following chapter and appendix.

Chapter 8 summarises the results and gives some directions for further research.

Appendix A gives the full source code of the Monatron library.

Related work is discussed in the summary at the end of each chapter.

1.4 Contributions

The primary contributions of the thesis are as follows:

- An abstract theory of lifting for monoid transformers, where monoids are taken in an unspecified monoidal category. This generalizes, extends and clarifies the incremental approach to modular monadic semantics. The abstract category-theoretic formulation of the theory opens the door to its application to other structures proposed for modelling computational effects, such as arrows (Hughes 2000) and Freyd categories (Power and Robinson 1997), which can be viewed as monoids in suitable monoidal categories (Heunen and Jacobs 2006) (Chapter 4).
- An algebraic presentation of the callcc operation, showing that it is well-behaved and that it can be lifted along any monad morphism (Chapter 3).
- Theory is put into practice with the implementation of a monad transformer library for Haskell which incorporates uniform liftings. Additionally, the traditional design of monad transformers libraries is refined with some conceptual changes (Chapter 5).
- An implementation of Turi's semantics-based approach to operational semantics. This implementation allows programmers to write small-step operational semantics in a natural manner, while keeping the benefits of denotational semantics. In fact operational semantics written in this way induce both a well-behaved transition system (where bisimulation is a congruence) and an internally fully-abstract denotational semantics. The ability to write well-behaved small-step semantics is particularly relevant to the semantics of concurrent processes. (Chapter 7).

- The aforementioned approach to operational semantics is extended with a technique for writing operational semantics modularly. The technique incorporates the current technology for modular monadic semantics to structure the functor representing observable behaviour (Chapter 7).

The source code for the chapters in the second part of the thesis can be downloaded from the author's webpage (`http://www.cs.nott.ac.uk/~mjj/thesis/`).

Published contributions

Some of the results in this thesis have been published by the author, are accepted for publication or have been submitted for publication.

- Mauro Jaskelioff and Eugenio Moggi (2009). "Monad Transformers as Monoid Transformers". In: *Theoretical Computer Science*. Submitted for publication.

  This article is the core of Part I. The idea of working with monoids in a monoidal category, and of considering monoidal functors as another interesting class of transformers is due to Moggi.

- Mauro Jaskelioff (2009). "Modular Monad Transformers". In: *European Symposium on Programming*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, pp. 64–79.

  This article develops the theory in System $F\omega$ and has been superseded by the more abstract category-theoretic presentation of the theory in Jaskelioff and Moggi (2009). However, its formulation of standard callcc in terms of algebraic operations is included in Chapter 3. In this paper only $\beta\eta$-equivalence is erroneously assumed, but actually a stronger equational theory is needed for the results to hold.

- Mauro Jaskelioff (2008). "Monatron: an Extensible Monad Transformer Library". In: *Implementation and Application of Functional Languages*. Accepted for publication.

  The main ideas of the Monatron library are presented in this article, but the library presented in Chapter 5 and Appendix A has been extended and simplified.

- Mauro Jaskelioff, Neil Ghani, and Graham Hutton (2008). "Modularity and Implementation of Mathematical Operational Semantics". In: *Proceedings of the Workshop on Mathematically Structured Functional Programming*. Reykjavik, Iceland.

  The results in this article have been refined in Chapter 7 and the main example has been extended with a parallel construct. The implementation of modular syntax in Haskell is presented in Chapter 6.

Prerequisites

For the first part of the thesis, a modest knowledge of Category Theory is assumed. The category-theoretic notions which are relevant to the thesis, but might be out of the scope of an introductory textbook, are recalled in Chapter 2. Further information can be found in more advanced textbooks such as Mac Lane (1971), Borceux (1994*a*,*b*), or Barr and Wells (1985, 1995).

For the second part, knowledge of Haskell (Peyton Jones 2003) is assumed. Readers unfamiliar with Haskell may consult introductory textbooks such as Hutton (2007), Thompson (1999), or Bird (1998).

Part I

Theory

Chapter 2

Categorical Background

It is well-known (Mac Lane 1971) that monads on a category $\mathcal{C}$ *correspond* to monoids in the (strict) monoidal category $\mathsf{Endo}(\mathcal{C})$ of endofunctors on $\mathcal{C}$. A similar correspondence holds when monads are replaced by *strong monads* on a cartesian closed category $\mathcal{C}$ or by *monads expressible* in system $F\omega$ (or some other typed calculus of adequate expressivity), provided $\mathsf{Endo}(\mathcal{C})$ is replaced with a suitable (strict) monoidal category $\hat{\mathcal{E}}$. These observations suggest that a theory of monad transformers can be viewed as an instance of a more abstract theory of *monoid transformers* in the setting of a monoidal category $\hat{\mathcal{E}}$.

Consequently, in this chapter, several categorical concepts which are needed to develop the theory in this abstract manner are introduced. First, *monoidal categories* (together with functors and natural transformations) are defined, along with other notions, such as exponentials and monoids, definable in the setting of any monoidal category. After these notions are introduced, several examples are given.

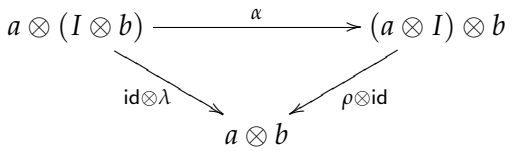
As a convention, a notion X with additional structure is written as $\hat{X}$.

2.1 Monoidal Categories

Definition 2.1 (Monoidal Category)**.** A *monoidal category* $\hat{\mathcal{E}}$ is a tuple $(\mathcal{E}, \otimes, I, \alpha, \lambda, \rho)$, where

- $\mathcal{E}$ is a category, $\otimes : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ is a bifunctor, $I \in \mathcal{E}$ is an object;
- $\alpha_{a,b,c} : a \otimes (b \otimes c) \to (a \otimes b) \otimes c$, $\lambda_a : I \otimes a \to a$, and $\rho_a : a \otimes I \to a$ are natural isomorphisms such that $\lambda_I = \rho_I$ and the following diagrams commute

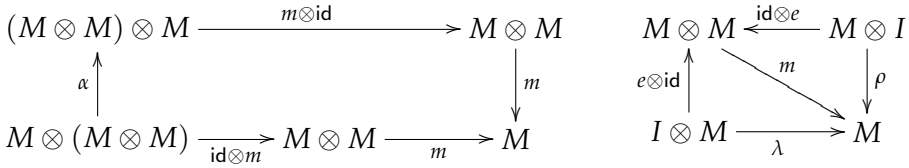
$$\begin{array}{ccccc} a \otimes (b \otimes (c \otimes d)) & \xrightarrow{\alpha} & (a \otimes b) \otimes (c \otimes d) & \xrightarrow{\alpha} & ((a \otimes b) \otimes c) \otimes d \\ \downarrow{\scriptstyle \mathsf{id} \otimes \alpha} & & & & \uparrow{\scriptstyle \alpha \otimes \mathsf{id}} \\ a \otimes ((b \otimes c) \otimes d) & & \xrightarrow[\alpha]{} & & (a \otimes (b \otimes c)) \otimes d \end{array}$$



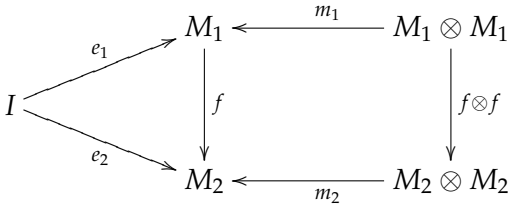
When the natural isomorphisms α, λ and ρ are identities, the diagrams necessarily commute, and the monoidal category is called *strict*. Also, when there is a natural isomorphism $\gamma_{a,b} : a \otimes b \to b \otimes a$ subject to some coherence conditions, the monoidal category is called *symmetric*.

Definition 2.2 (Monoid)**.** The category $\mathsf{Mon}(\hat{\mathcal{E}})$ of monoids in a monoidal category $\hat{\mathcal{E}}$ is given by

objects are *monoids* $\hat{M} = (M, e, m)$, i.e. diagrams $I \xrightarrow{e} M \xleftarrow{m} M \otimes M$ in $\mathcal{E}$ such that



arrows from $\hat{M}_1$ to $\hat{M}_2$ are arrows $M_1 \xrightarrow{f} M_2$ in $\mathcal{E}$ such that

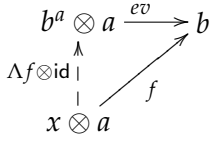


Identities and composition in $\mathsf{Mon}(\hat{\mathcal{E}})$ are inherited from $\mathcal{E}$.

The forgetful functor $U : \mathsf{Mon}(\hat{\mathcal{E}}) \longrightarrow \mathcal{E}$ maps a monoid $\hat{M}$ to M and an arrow $\hat{M}_1 \xrightarrow{f} \hat{M}_2$ to $M_1 \xrightarrow{f} M_2$.

Definition 2.3 (Exponential)**.** An *exponential* of b to a in a monoidal category $\hat{\mathcal{E}}$ is an object b^a together with a map $ev : b^a \otimes a \longrightarrow b$ satisfying the universal property

$$\forall x \in \mathcal{E}.\, \forall f : x \otimes a \longrightarrow b.\, \exists!\, \Lambda f : x \dashrightarrow b^a \text{ such that}$$



Definition 2.4 (Monoidal Right-Closed Category)**.** A *monoidal right-closed category* is a monoidal category $\hat{\mathcal{E}}$ with exponentials. That is, for every object a and b, the exponential b^a exists.

Equivalently, a monoidal category is right-closed when, for every object a, the functor $- \otimes a$ has a right adjoint, i.e. for every a there is an isomorphism natural in x and b:

$$\hat{\mathcal{E}}(x \otimes a, b) \cong \hat{\mathcal{E}}(x, b^a)$$

By considering right adjoints to left tensoring one obtains the notion of *monoidal left-closed category*. When $\hat{\mathcal{E}}$ is symmetric monoidal, one simply speaks of *monoidal closed category*, since the category is left-closed if and only if it is right-closed.

Definition 2.5 (Monoidal Functor)**.** Given two monoidal categories $\hat{\mathcal{E}}$ and $\hat{\mathcal{E}}'$, a *monoidal functor* $\hat{T}$ from $\hat{\mathcal{E}}$ to $\hat{\mathcal{E}}'$ is a tuple (T, ϕ_I, ϕ), where

- $T : \mathcal{E} \longrightarrow \mathcal{E}'$ is a functor
- $\phi_I : I' \longrightarrow TI$ is a map, $\phi_{a,b} : Ta \otimes' Tb \longrightarrow T(a \otimes b)$ is a natural transformation such that

$$\begin{array}{ccccc} Ta \otimes' (Tb \otimes' Tc) & \xrightarrow{\mathsf{id}\otimes'\phi} & Ta \otimes' T(b \otimes c) & \xrightarrow{\phi} & T(a \otimes (b \otimes c)) \\ \downarrow{\scriptstyle \alpha'} & & & & \downarrow{\scriptstyle T(\alpha)} \\ (Ta \otimes' Tb) \otimes' Tc & \xrightarrow[\phi\otimes'\mathsf{id}]{} & T(a \otimes b) \otimes' Tc & \xrightarrow[\phi]{} & T((a \otimes b) \otimes c) \end{array}$$

$$\begin{array}{ccc} I' \otimes' Ta & \xrightarrow{\lambda'} & Ta \\ \downarrow{\scriptstyle \phi_I\otimes'\mathsf{id}} & & \uparrow{\scriptstyle T(\lambda)} \\ TI \otimes' Ta & \xrightarrow[\phi]{} & T(I \otimes a) \end{array} \qquad \begin{array}{ccc} Ta \otimes' I' & \xrightarrow{\rho'} & Ta \\ \downarrow{\scriptstyle \mathsf{id}\otimes'\phi_I} & & \uparrow{\scriptstyle T(\rho)} \\ Ta \otimes' TI & \xrightarrow[\phi]{} & T(a \otimes I) \end{array}$$

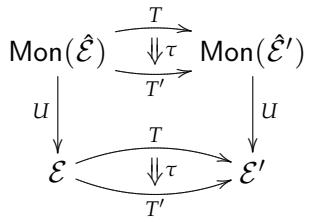
When the map ϕ_I and the natural transformation ϕ are identities, the monoidal functor is called *strict*, and the commuting diagrams amount to require that $I' = TI$, $Ta \otimes' Tb = T(a \otimes b)$, $\alpha' = T(\alpha)$, $\lambda' = T(\lambda)$, and $\rho' = T(\rho)$. Monoidal functors (as defined above) do not require ϕ_I and ϕ to be isomorphisms. Some authors call these functors *lax monoidal functors* and reserve the term monoidal functor for the case when ϕ_I and ϕ are isomorphisms.

Definition 2.6 (Monoidal Natural Transformation)**.** Given the monoidal functors $\hat{T}$ and $\hat{T}'$ from $\hat{\mathcal{E}}$ to $\hat{\mathcal{E}}'$, a *monoidal natural transformation* τ from $\hat{T}$ to $\hat{T}'$ is a natural transformation $\tau : T \xrightarrow{\bullet} T'$ such that

$$\begin{array}{ccc} & I' & \\ {\scriptstyle \phi_I}\swarrow & & \searrow{\scriptstyle \phi'_I} \\ TI & \xrightarrow[\tau]{} & T'I \end{array} \qquad \begin{array}{ccc} Ta \otimes' Tb & \xrightarrow{\tau\otimes'\tau} & T'a \otimes' T'b \\ \downarrow{\scriptstyle \phi} & & \downarrow{\scriptstyle \phi'} \\ T(a \otimes b) & \xrightarrow[\tau]{} & T'(a \otimes b) \end{array}$$

The following theorem shows that monoidal functors and monoidal natural transformations extend to the category $\mathsf{Mon}(\hat{\mathcal{E}})$ of monoids in the monoidal category $\hat{\mathcal{E}}$.

Theorem 2.7 (Monoidal Extension)**.** *A monoidal functor $\hat{T} : \hat{\mathcal{E}} \longrightarrow \hat{\mathcal{E}}'$ induces a functor $T : \mathsf{Mon}(\hat{\mathcal{E}}) \longrightarrow \mathsf{Mon}(\hat{\mathcal{E}}')$. Similarly a monoidal natural transformation $\tau : \hat{T} \xrightarrow{\bullet} \hat{T}'$ induces a natural transformation $\tau : T \xrightarrow{\bullet} T'$ such that*

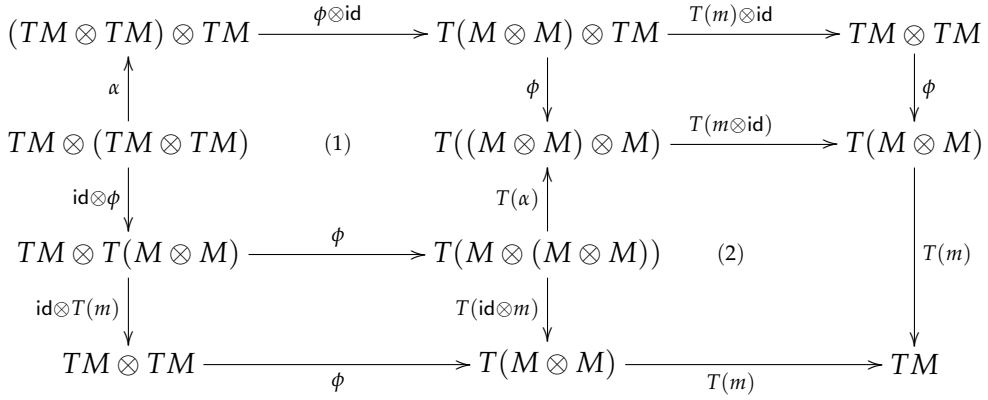


The induced functor and natural transformation are:

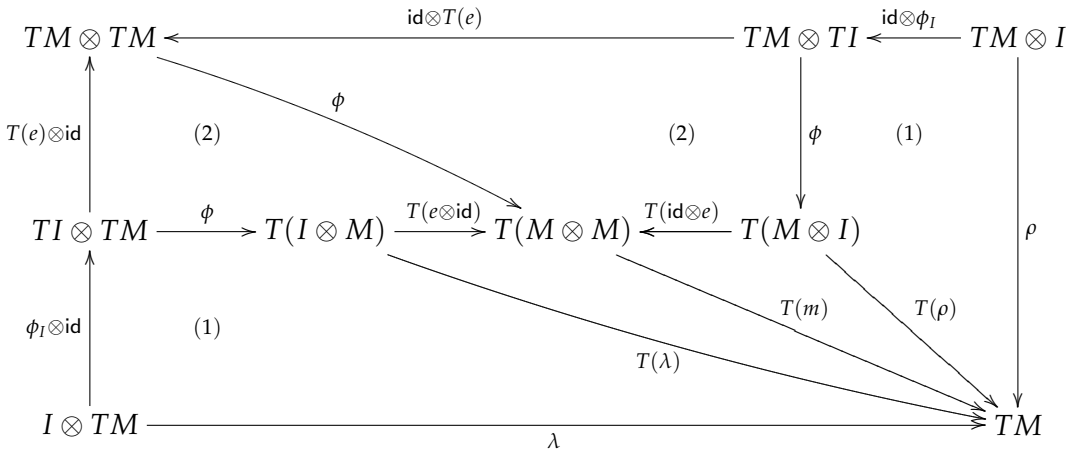
$$T\hat{M} \;\hat{=}\; I' \xrightarrow{\phi_I} TI \xrightarrow{T(e)} TM \xleftarrow{T(m)} T(M \otimes M) \xleftarrow{\phi} TM \otimes' TM$$

$$\tau_{\hat{M}} \;\hat{=}\; T\hat{M} \xrightarrow{\tau_M} T'\hat{M}$$

Proof. We first prove that $T\hat{M}$ is a monoid.

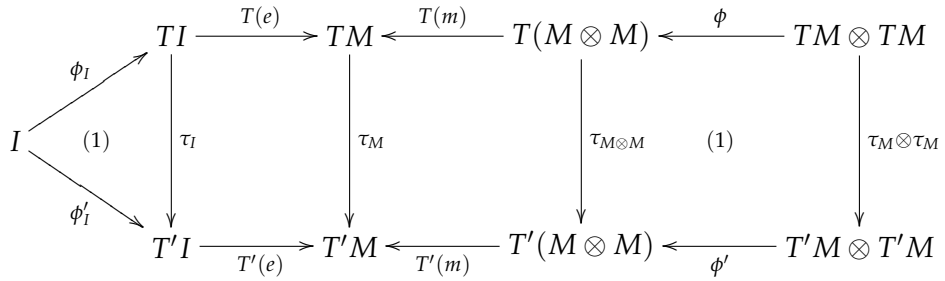


The diagram commutes, (1) because T is monoidal, (2) because $\hat{M}$ is a monoid (and functoriality of T), and the rest by naturality of ϕ.



The diagram commutes, (1) because T is monoidal, (2) by naturality of ϕ, and the rest because $\hat{M}$ is a monoid (and functoriality of T).

The following diagram commutes, (1) because τ is a monoidal natural transformation and the rest by naturality of τ, therefore proving that $\tau_{\hat{M}}$ is a monoid homomorphism.



□

2.2 Examples of Monoidal Categories

We give several examples of monoidal categories. The definition of monoidal category is *self-dual*, i.e. when $\mathcal{E}$ is monoidal, then $\mathcal{E}^{op}$ is monoidal as well. Therefore, each of these examples has a dual.

Example 2.8. A category $\mathcal{C}$ with **finite products** (e.g. the category **Set** of sets) forms a symmetric monoidal category $(\mathcal{C}, \times, 1, \alpha, \lambda, \rho)$, where $\times$ is a binary product functor, 1 is a terminal object, and the natural isomorphisms are uniquely determined by the universal properties of products. The category is monoidal closed and exponentials (in the sense of Definition 2.3) correspond to the usual notion of exponentials for a cartesian closed category.

Example 2.9. If $\mathcal{C}$ is a (small) category, then the category $\mathsf{Endo}(\mathcal{C})$ of **endofunctors** over $\mathcal{C}$ forms a strict monoidal category $(\mathsf{Endo}(\mathcal{C}), \circ, \mathsf{Id})$, where $\circ$ is functor composition and Id is the identity functor. More precisely,

objects are endofunctors $F : \mathcal{C} \longrightarrow \mathcal{C}$;

arrows from F to G are natural transformations $\tau : F \xrightarrow{\bullet} G$;

tensor is functor composition $(G \circ F)(-) \mathrel{\hat{=}} G(F(-))$;

unit is the identity functor $\mathsf{Id}(-) \mathrel{\hat{=}} -$.

Also the category of **profunctors** $\mathcal{C}^{op} \times \mathcal{C} \longrightarrow \mathbf{Set}$ forms a monoidal category (see Borceux 1994*a*), and there is a monoidal functor from endofunctors to profunctors mapping F to $\mathcal{C}(-_1, F-_2)$.

If $\mathcal{C}$ has *J-limits*, i.e. limits for diagrams of shape J, then so does $\mathsf{Endo}(\mathcal{C})$. These J-limits in $\mathsf{Endo}(\mathcal{C})$ are computed pointwise and are preserved by pre-composition of functors,

i.e. functors $- \circ F : \mathsf{Endo}(\mathcal{C}) \longrightarrow \mathsf{Endo}(\mathcal{C})$ (J-colimits are also computed pointwised and preserved by pre-composition of functors).

In this monoidal category, an exponential G^F corresponds to a right Kan's extension of G along F, characterized by a bijection from $H \xrightarrow{\bullet} G^F$ to $H \circ F \xrightarrow{\bullet} G$ natural in the endofunctor H.

A category $\mathcal{C}$ is *locally finitely presentable* iff it is cocomplete, and has a strong generator of *finitely presentable* objects (objects X of $\mathcal{C}$ such that its hom-functor $\mathcal{C}(X, -) : \mathcal{C} \to \mathbf{Set}$ preserves filtered colimits). For example, **Set** is locally finitely presentable. In general, the category $\mathsf{Endo}(\mathcal{C})$ is not right-closed, but the full subcategory category $\mathsf{Endo}(\mathcal{C})_f$ of *finitary* endofunctors (i.e. endofunctors preserving filtered colimits) over a locally finitely presentable category $\mathcal{C}$ has a right-closed monoidal structure (Kelly and Power 1993).

Example 2.10 (due to Eugenio Moggi)**.** Let $(A, \cdot)$ be a *partial combinatory algebra*, i.e. a set A with two distinct elements $\mathsf{K} \neq \mathsf{S}$ and a partial binary operation $\cdot : A \times A \rightharpoonup A$, we write $a\ b$ for $\cdot(a, b)$, such that

$\mathsf{K}\,x\,y = x$	i.e. $\mathsf{K}\,x\,y$ is defined and equal to x
$\mathsf{S}\,x\,y \downarrow$	i.e. $\mathsf{S}\,x$ and $(\mathsf{S}\,x)\,y$ are defined
$\mathsf{S}\,x\,y\,z \simeq x\,z\,(y\,z)$	i.e. both terms are either undefined or equal

The category $\mathcal{P}_A$ of **partial equivalence relations** over A is given by

objects are symmetric and transitive relations $R \subseteq A \times A$ (called PERs); A/R denotes the set of R-equivalence classes, i.e. the set of subsets $X \subseteq A$ such that $\exists x \in X \wedge (\forall a \in A.\, a \in X \iff aRx)$;

arrows from R_1 to R_2 are maps $f : A/R_1 \longrightarrow A/R_2$ with a **realiser**, i.e. an $r \in A$ such that $\forall X \in A/R_1.\, \forall x \in X.\, r\,x \in f(X)$ ($r \vdash_A f$ for short).

The fact that $(A, \cdot)$ is a partial combinatory algebra ensures that identity maps are realisable, composition of realisable maps is realisable, $\mathcal{P}_A$ is *locally cartesian closed* and has finite colimits.

Consider the strict monoidal category $\mathsf{Endo}(\mathcal{P}_A)$. It has a proper sub-category $\mathsf{Endo}(\mathcal{P}_A)_r$ of **realisable endofunctors** and realisable natural transformations given by:

objects are endofunctors $F : \mathcal{P}_A \longrightarrow \mathcal{P}_A$ with a **realiser**, i.e. an $r \in A$ such that $a \vdash_A f$ implies $r\,a \vdash_A F(f)$ for every $a \in A$ and arrow f in $\mathcal{P}_A$.

arrows from F to G are natural transformations $\tau : F \xrightarrow{\bullet} G$ with a **realiser**, i.e. an $r \in A$ such that $r \vdash_A \tau_R$ for every object R of $\mathcal{P}_A$.

The category $\mathsf{Endo}(\mathcal{P}_A)_r$ inherits the strict monoidal structure of $\mathsf{Endo}(\mathcal{P}_A)$, because realisable endofunctors and realisable natural transformations are closed w.r.t. identities and

composition. Therefore the inclusion of $\mathsf{Endo}(\mathcal{P}_A)_r$ into $\mathsf{Endo}(\mathcal{P}_A)$ is a strict monoidal functor. A remarkable property of $\mathsf{Endo}(\mathcal{P}_A)_r$, not shared by $\mathsf{Endo}(\mathcal{P}_A)$, is that it is right-closed. The exponential G^F for any pair of realisable functors F and G is constructed in the following way:

- $a\ G^F R\ b \iff a$ and b are realisers for the same realisable natural transformation $\tau : Y_R \otimes F \xrightarrow{\bullet} G$, where Y_R is the realisable endofunctor $Y_R(-) \mathrel{\hat{=}} -^R$ given by exponentiation to R in $\mathcal{P}_A$.
- An arrow $R \xrightarrow{f} S$ in $\mathcal{P}_A$ induces a realisable natural transformation $Y(f) : Y_S \xrightarrow{\bullet} Y_R$ such that $Y(f)_T \mathrel{\hat{=}} T^f$. Therefore, when the arrow $Y_R \otimes F \xrightarrow{\tau} G$ is realisable, then the arrow $Y_S \otimes F \xrightarrow{Y(f)\otimes \mathsf{id}_F} Y_R \otimes F \xrightarrow{\tau} G$ is realisable as well. This induces a function $G^F(f) : A/G^F(R) \longrightarrow A/G^F(S)$, and by elementary considerations one can give an $a \in A$ such that $a\, r \vdash_A G^F(f)$ whenever $r \vdash_A f$.

Example 2.11. Consider system F (Girard 1972; Reynolds 1974) (also known as the polymorphic lambda calculus). Bainbridge et al. (1990) showed that mixed-variant functors over $\mathcal{P}_A$ (with $\mathcal{P}_A$ as in the previous example) and dinaturals transformations form a category that provides a parametric model of system F without ad-hoc functions. We identify terms that are equal in the model and define the strict monoidal category $\hat{\mathcal{E}}_F$ of endofunctors and natural transformations **expressible** in system F (Reynolds and Plotkin 1993). To fix the notation, we recall the syntax

$$\begin{array}{ll} \text{types} & U ::= X \mid U \to U \mid \forall X.\, U \\ \text{terms} & e ::= x \mid \lambda x : U.\, e \mid e\, e \mid \Lambda X.\, e \mid e\, U \end{array}$$

and some notational conventions: we write e_U for $e\, U$ (polymorphic instantiation) and we write definitions $g_X(x : A) \mathrel{\hat{=}} t$ for $g \mathrel{\hat{=}} \Lambda X.\, \lambda x : A.\, t$.

objects are *expressible endofunctors*, i.e. pairs $\hat{F} = (F[-], \mathsf{map}^F)$ with $F[-]$ a type scheme and $\mathsf{map}^F : \forall X, Y.\, (X \to Y) \to F[X] \to F[Y]$ a closed term such that the following equivalences hold.

$$\begin{aligned} \mathsf{map}^F_{A,A}\, \mathsf{id}_A &= \mathsf{id}_{F[A]} \\ \mathsf{map}^F_{A,C}\, (g \cdot f) &= \mathsf{map}^F_{B,C}\, g \cdot \mathsf{map}^F_{A,B}\, f \end{aligned}$$

where, $\mathsf{id}_X \mathrel{\hat{=}} \lambda x : X.\, x$ is the identity on X and $g \cdot f \mathrel{\hat{=}} \lambda x : X.\, g\,(f\, x)$ is the composition of $g : Y \to Z$ and $f : X \to Y$.

arrows from $\hat{F}$ to $\hat{G}$ are *expressible natural transformations*, i.e. closed terms $\tau : \forall X.\, F[X] \to G[X]$. The interpretation of $\forall$ in the model as a *realisable end* ensures that the naturality condition holds for τ:

$$\mathsf{map}^G_{A,B}\, f \cdot \tau_A \;=\; \tau_B \cdot \mathsf{map}^F_{A,B}\, f$$

Identity on $\hat{F}$ is $\iota_F \mathrel{\hat{=}} \Lambda X.\, \lambda x : F[X].\, x$, and composition of σ and τ is $\sigma \circ \tau \mathrel{\hat{=}} \Lambda X.\, \sigma_X \cdot \tau_X$. The monoidal structure is given by:

tensor $\hat{F} \circ \hat{G}$ is $(F[G[-]], \mathsf{map})$ with $\mathsf{map}_{A,B}\,(f : A \to B) \mathrel{\hat{=}} \mathsf{map}^F_{G[A],G[B]}\,(\mathsf{map}^G_{A,B}\, f)$

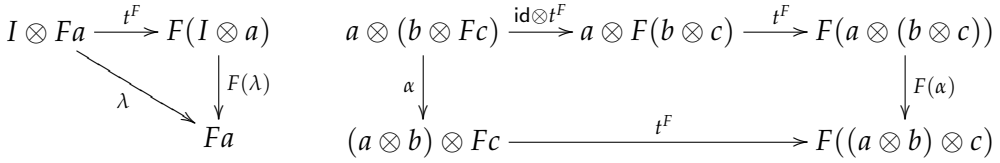
unit is the pair $([-], \mathsf{map})$ with $\mathsf{map}_{A,B}\,(f : A \to B) \mathrel{\hat{=}} f$.

This monoidal category is right-closed. The exponential for $\hat{G}$ to $\hat{F}$ is the expressible endofunctor $\hat{H}$ given by

$$H[X] = \forall Z.\, (X \to F[Z]) \to G[Z]$$
$$\mathsf{map}^H_{X,Y}\,(f : X \to Y, h : H[X]) \mathrel{\hat{=}} \Lambda Z.\, \lambda g : Y \to F[Z].\, h_Z\,(g \cdot f)$$

Example 2.12. If $\hat{\mathcal{E}}$ is a (small) monoidal category, then the category $\mathsf{Endo}(\hat{\mathcal{E}})_s$ of **strong endofunctors** over $\hat{\mathcal{E}}$ forms a strict monoidal category $(\mathsf{Endo}(\hat{\mathcal{E}})_s, \circ, \hat{\mathsf{Id}})$, more precisely

objects are $\hat{F} = (F, t^F)$ with $F : \mathcal{E} \longrightarrow \mathcal{E}$ functor $t^F_{a,b} : a \otimes Fb \longrightarrow F(a \otimes b)$ natural transformation such that



arrows from $\hat{F}$ to $\hat{G}$ are natural transformations $\tau : F \xrightarrow{\bullet} G$ such that

$$\begin{array}{ccc} a \otimes Fb & \xrightarrow{\mathsf{id} \otimes \tau} & a \otimes Gb \\ {\scriptstyle t^F}\downarrow & & \downarrow{\scriptstyle t^G} \\ F(a \otimes b) & \xrightarrow[\tau]{} & G(a \otimes b) \end{array}$$

tensor $\hat{G} \circ \hat{F}$ is the pair $(G \circ F, t)$ with

$$t_{a,b} \mathrel{\hat{=}}\ a \otimes G(Fb) \xrightarrow{t^G} G(a \otimes Fb) \xrightarrow{G(t^F)} G(F(a \otimes b))$$

unit $\hat{\mathsf{Id}}$ is the pair (Id, t) with $t_{a,b} \mathrel{\hat{=}} \mathsf{id}_{a \otimes b}$.

Moreover, the forgetful functor $U : \mathsf{Endo}(\hat{\mathcal{E}})_s \longrightarrow \mathsf{Endo}(\mathcal{E})$, mapping $\hat{F}$ to F, is strict monoidal. Also the category $\mathsf{Endo}(\hat{\mathcal{E}})_m$ of **monoidal endofunctors** forms a strict monoidal category.

Example 2.13. Given a monoidal category $\hat{\mathcal{E}}$ with J-limits, i.e. limits for diagrams of shape J, we write $\mathsf{Lim}_J(\hat{\mathcal{E}})$ for the full sub-category of $\mathcal{E}$ whose objects $a \in \mathcal{E}$ **preserve J-limits**, i.e. the functor $a \otimes - : \mathcal{E} \longrightarrow \mathcal{E}$ preserves J-limits. This sub-category inherits the monoidal structure from $\mathcal{E}$, in fact

- I preserves J-limits, because $I \otimes -$ is isomorphic (through λ) to the identity functor on $\mathcal{E}$, and the identity functor preserves all limits;
- if a and b preserve J-limits, then so does $a \otimes b$, because $(a \otimes b) \otimes -$ is isomorphic (through α) to $a \otimes (b \otimes -)$, which is the composition of the J-limits preserving functors $a \otimes -$ and $b \otimes -$.

When $\mathcal{C}$ is a (small) category with J-limits, then the (strict) monoidal category $\hat{\mathcal{E}}$ of endofunctors over $\mathcal{C}$ has J-limits (see Example 2.9), and $\mathsf{Lim}_J(\hat{\mathcal{E}})$ is exactly the category of endofunctors on $\mathcal{C}$ preserving J-limits in $\mathcal{C}$.

2.3 Examples of Monoids

In this section, constructions of objects in $\mathsf{Mon}(\hat{\mathcal{E}})$ are given. The constructions may require additional assumptions on the monoidal category $\hat{\mathcal{E}}$. More examples of monoids, in the form of strong monads, will be given in Section 3.2.

Example 2.14. The **initial monoid** $\hat{I}$, is given by the diagram

$$I \xrightarrow{\mathsf{id}} I \xleftarrow{\lambda} I \otimes I$$

In fact, $\hat{I}$ is an initial object in $\mathsf{Mon}(\hat{\mathcal{E}})$.

Example 2.15. When $\mathcal{E}$ has J-limits, then $\mathsf{Mon}(\hat{\mathcal{E}})$ has J-limits which are computed pointwise, therefore they are preserved by the forgetful functor U. In particular, if $\mathcal{E}$ has a terminal object 1, then on 1 there is a unique monoid structure $\hat{1}$, which yields a terminal object in $\mathsf{Mon}(\hat{\mathcal{E}})$.

Example 2.16. When the exponential a^a exists, we have a **monoid of endomorphisms** on a, given by the diagram

$$I \xrightarrow{i_a} a^a \xleftarrow{c_a} a^a \otimes a^a \quad \text{where}$$

$$\begin{aligned} i_a &\triangleq \Lambda(I \otimes a \xrightarrow{\lambda} a) \\ c_a &\triangleq \Lambda\big((a^a \otimes a^a) \otimes a \xrightarrow{\alpha^{-1}} a^a \otimes (a^a \otimes a) \xrightarrow{\mathsf{id} \otimes ev} a^a \otimes a \xrightarrow{ev} a\big) \end{aligned}$$

Moreover, if $\hat{M} = (M, e, m)$ is a monoid, then $M \xrightarrow{\Lambda m} M^M$ is a monoid morphism from $\hat{M}$ to the monoid of endomorphisms on M.

Example 2.17. When the left-adjoint $(-)^*$ to $U : \mathsf{Mon}(\hat{\mathcal{E}}) \longrightarrow \mathcal{E}$ exists, it gives **free monoids**. Sufficient conditions for the existence of free monoids are:

- The category $\mathcal{E}$ has denumerable coproducts, and for each $a \in \mathcal{E}$ the functors $a \otimes -$ and $- \otimes a$ preserve these coproducts. The free monoid on a exists (see Mac Lane 1971, XII.Thm 2) and its carrier is given by the coproduct of the family $(a^n \mid n \in N)$ with $a^0 \triangleq I$ and $a^{n+1} \triangleq a \otimes a^n$.

- The category $\mathcal{E}$ has binary coproducts and ω-colimits, and for each $a \in \mathcal{E}$ the functors $a \otimes -$ and $- \otimes a$ preserves ω-colimits and $- \otimes a$ preserves also binary coproducts. The free monoid a^* exists (see Kelly 1980, Section 23; or, alternatively, Rezk 1996, Appendix A) and its carrier is given by the colimit of the ω-chain $(a_n \xrightarrow{f_n} a_{n+1} \mid n \in N)$, where

$$f_0 \triangleq I \xrightarrow{\mathsf{inl}} I + (a \otimes I)$$
$$f_{n+1} \triangleq I + (a \otimes a_n) \xrightarrow{\mathsf{id}+(\mathsf{id}\otimes f_n)} I + (a \otimes a_{n+1}).$$

If $\hat{\mathcal{E}}$ is right-closed, then $- \otimes a$ will be a left adjoint and therefore it will preserve all colimits. Also, as mentioned in Example 2.9, if $\hat{\mathcal{E}}$ is an endofunctor category, $- \otimes a$ will also preserve colimits.

2.4 Summary

We have presented the notions of monoids in a monoidal category, as a generalization of the application we have in mind, which is the modelling of computational effects by monads. The advantage of working with monoids in a monoidal category is that monoids are simpler, and that by working at that level of generality other structures which are used for modelling effects can be incorporated into the theory. For example, arrows (Hughes 2000) and Freyd categories (Power and Robinson 1997) can be viewed as monoids in suitable monoidal categories (Heunen and Jacobs 2006).

Many textbooks cover the subject of monoidal categories. Two standard references are Mac Lane (1971), and Borceux (1994*b*).

Several examples of monoidal categories were introduced in this chapter. The category of finitary endofunctors $\mathsf{Endo}(\mathcal{C})_f$ over a locally finitely presentable category $\mathcal{C}$ (Example 2.9), the category of realisable endofunctors $\mathsf{Endo}(\mathcal{P}_A)_r$ (Example 2.10) and the category of expressible endofunctors $\hat{\mathcal{E}}_F$ (Example 2.11) have the remarkable characteristic of being right-closed. This will play an important role in one of the liftings obtained in Chapter 4. For more information on finitary endofunctors and locally finitely presentable categories, see Adámek and Rosický (1994), for more information on realisable endofunctors see Hyland (1988), or Asperti and Longo (1990).

Chapter 3

Operations and Lifting

In this chapter, the notions of operation on a monoid and lifting of operations are introduced. A classification of operations into *H-operations*, *first-order operations* and *algebraic operations* is given, and it is shown that for every algebraic operation a lifting exists and is unique. In the following chapter, we establish lifting results for wider classes of operations.

We give several examples of monoids in $\mathsf{Endo}(\mathbf{Set})$ (i.e. monads on **Set**) that model computational effects and present some of the effect-manipulating operations that are usually associated to them.

3.1 Abstract Operations and Algebraic Lifting

We will work with the following notions of operation.

Definition 3.1 (Operations)**.** Given a functor $H : \mathsf{Mon}(\hat{\mathcal{E}}) \longrightarrow \mathcal{E}$, an *H-operation* for the monoid $\hat{M} = (M, e, m)$ is a map $H\hat{M} \xrightarrow{op} M$ in $\mathcal{E}$.

A *first-order operation* for $\hat{M}$ of signature $S \in \mathcal{E}$ is a map $S \otimes M \xrightarrow{op} M$, i.e. op is an H-operation for the functor $H(-) = S \otimes U(-)$, and such op is called *algebraic* when the following diagram commutes

$$\begin{array}{ccccc} S \otimes (M \otimes M) & \xrightarrow{\alpha} & (S \otimes M) \otimes M & \xrightarrow{op \otimes \mathsf{id}} & M \otimes M \\ {\scriptstyle \mathsf{id} \otimes m}\downarrow & & & & \downarrow{\scriptstyle m} \\ S \otimes M & & \xrightarrow[op]{} & & M \end{array}$$

Algebraic operations as presented here are a generalization of the standard notion of algebraic operation (Plotkin and Power 2001*b*, see also remark 3.4). The terminology for H-operation and first-order operation comes from Jaskelioff (2009), where the former corresponds to a higher-order functor, and the latter is composition with a first-order functor.

An operation associated to a monoid $\hat{M}$ may fail to be an H-operation, but can often be *defined* from one, and we say that these operations are *derived* from the H-operations.

For instance, when $\hat{\mathcal{E}}$ has the dual of exponentials $G \xrightarrow{\overline{ev}} G_F \otimes F$ (see Definition 2.3), then there is a bijection between operations $H'\hat{M} \longrightarrow M \otimes F$ and H-operations $H\hat{M} \longrightarrow M$ with $H(-) \triangleq (H'-)_F$. Other instances of derived operations will be shown in the next section.

Given two H-operations on the same functor H but on different monoids, we say that one is a lifting of the other, when they are "related" by a monoid morphism. The following definition makes this precise.

Definition 3.2 (Lifting)**.** Given an H-operation $H\hat{M} \xrightarrow{op^M} M$ for $\hat{M}$ and a monoid morphism $h : \hat{M} \longrightarrow \hat{N}$, an H-operation $H\hat{N} \xrightarrow{op^N} N$ for $\hat{N}$ is a *lifting of op along h* when

$$\begin{array}{ccc} H\hat{N} & \xrightarrow{op^N} & N \\ {\scriptstyle H(h)}\uparrow & & \uparrow{\scriptstyle h} \\ H\hat{M} & \xrightarrow[op^M]{} & M \end{array}$$

Given an operation op on a monoid $\hat{M}$ and a monoid morphism $h : \hat{M} \longrightarrow \hat{N}$, the lifting problem consists of finding an operation op^N on $\hat{N}$ that is a lifting of op along h. In the following, we show that every algebraic operation lifts along any monoid morphism. First, we prove a bijection between algebraic operations and certain morphisms that is essential to proving the main theorem of this section, and that serves as an alternative characterization of algebraic operations.

Proposition 3.3. *Algebraic operations $S \otimes M \xrightarrow{op} M$ for $\hat{M} = (M, e, m)$ are in bijective correspondence with maps $S \xrightarrow{op'} M$, i.e. H-operations for the functor $H(-) = S$. The bijective correspondence is given by*

$$\begin{aligned} \phi(op) &\triangleq S \xrightarrow{\rho^{-1}} S \otimes I \xrightarrow{\mathsf{id}\otimes e} S \otimes M \xrightarrow{op} M \\ \psi(op') &\triangleq S \otimes M \xrightarrow{op'\otimes \mathsf{id}} M \otimes M \xrightarrow{m} M \end{aligned}$$

Proof. We calculate:

$$
\begin{aligned}
& \phi(\psi(op')) \\
= \quad & \{\text{Definition of } \phi \text{ and } \psi\} \\
& m \circ (op' \otimes \mathsf{id}) \circ (\mathsf{id} \otimes e) \circ \rho^{-1} \\
= \quad & \{\text{Bifunctoriality of } \otimes\} \\
& m \circ (\mathsf{id} \otimes e) \circ (op' \otimes \mathsf{id}) \circ \rho^{-1} \\
= \quad & \{\text{Monoid law}\} \\
& \rho \circ (op' \otimes \mathsf{id}) \circ \rho^{-1} \\
= \quad & \{\rho \text{ is a natural isomorphism}\} \\
& op'
\end{aligned}
$$

For the other direction, we calculate:

$$
\begin{aligned}
& \psi(\phi(op)) \\
= \quad & \{\text{Definition of } \phi \text{ and } \psi\} \\
& m \circ (op \otimes \mathsf{id}) \circ ((\mathsf{id} \otimes e) \otimes \mathsf{id}) \circ (\rho^{-1} \otimes \mathsf{id}) \\
= \quad & \{\alpha \text{ is a natural isomorphism}\} \\
& m \circ (op \otimes \mathsf{id}) \circ \alpha \circ (\mathsf{id} \otimes (e \otimes \mathsf{id})) \circ \alpha^{-1} \circ (\rho^{-1} \otimes \mathsf{id}) \\
= \quad & \{op \text{ is algebraic, monoidal isomorphisms}\} \\
& op \circ (\mathsf{id} \otimes m) \circ (\mathsf{id} \otimes (e \otimes \mathsf{id})) \circ (\mathsf{id} \otimes \lambda^{-1}) \\
= \quad & \{\text{Monoid law}\} \\
& op \circ (\mathsf{id} \otimes \lambda) \circ (\mathsf{id} \otimes \lambda^{-1}) \\
= \quad & \{\text{Isomorphism}\} \\
& op
\end{aligned}
$$

□

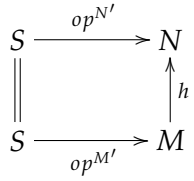
Remark 3.4. Monoids in the category of endofunctors on a cartesian closed category $\mathcal{C}$ are monads, and operations are natural transformations. When $SX = A \times X^B$, there is a further bijection between algebraic operations op for a monad $\hat{M}$ and maps $op'' : A \to MB$, namely

$$op''(a : A) \mathrel{\hat{=}} op_B(a, \mathsf{ret}_B^M).$$

where ret_B^M is the unit of the monad $\hat{M}$.

Theorem 3.5 (Unique algebraic lifting)**.** *If $op^M : S \otimes M \longrightarrow M$ is an algebraic operation for a monoid $\hat{M}$, and $h : \hat{M} \longrightarrow \hat{N}$ is a monoid morphism, then there is a unique $op^N : S \otimes N \longrightarrow N$ which is both algebraic for $\hat{N}$ and a lifting of op^M along h.*

Proof. Use Proposition 3.3 and replace op^M and op^N with $op^{M'}$ and $op^{N'}$. The following lifting is obtained:



□

3.2 Examples of Monads and Their Operations

In this section we give examples of (strong) monads on **Set** and associated operations, saying explicitly whether the operations are algebraic, first-order or more general instances of H-operations.

Monads on the cartesian closed category **Set** of sets coincide with strong monads, since every endofunctor on **Set** is strong, more precisely $U : \mathsf{Endo}(\mathbf{Set})_s \longrightarrow \mathsf{Endo}(\mathbf{Set})$ of Example 2.12 is an isomorphism. In other cartesian closed categories this is not the case, and the importance of monads being strong becomes more evident. For example, if a monad $\hat{M}$ is not strong, then a term $(\mathsf{let}_M\ x = e'\ \mathsf{in}\ e)$ is problematic when e has a free variable $y \neq x$ (for details, see Moggi 1989*b*, 1991).

There are equivalent ways of defining strong monads on a cartesian closed category $\mathcal{C}$, we borrow the definition adopted in Haskell, and freely use simply typed lambda-calculus as *internal language* to denote objects and maps in $\mathcal{C}$. When defining operations which are H-operations we omit the result type and freely use standard syntactic-sugar to make the definitions more clear.

Definition 3.6 (Strong Monad)**.** A *strong monad* on a cartesian closed category $\mathcal{C}$ is a triple $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$ consisting of

- a map $M : |\mathcal{C}| \longrightarrow |\mathcal{C}|$ on the objects of $\mathcal{C}$
- a family $\mathsf{ret}^M_X : X \longrightarrow MX$ of maps with $X \in \mathcal{C}$
- a family $\mathsf{bind}^M_{X,Y} : MX \times (MY)^X \longrightarrow MY$ of maps with $X, Y \in \mathcal{C}$

such that for every $a : A$, $f : (MB)^A$, $u : MA$ and $g : (MC)^B$

$$\begin{aligned} \mathsf{bind}^M_{A,B}(\mathsf{ret}^M_A(a), f) : MB &= f\ a \\ \mathsf{bind}^M_{A,A}(u, \mathsf{ret}^M_A) : MA &= u \\ \mathsf{bind}^M_{A,C}(u, \lambda a : A.\, \mathsf{bind}^M_{B,C}(f\ a, g)) : MC &= \mathsf{bind}^M_{B,C}(\mathsf{bind}^M_{A,B}(u, f), g) \end{aligned}$$

Definition 3.7 (Strong Monad Morphism)**.** A *strong monad morphism* $\tau : \hat{M} \longrightarrow \hat{N}$ is a family $\tau_X : MX \longrightarrow NX$ of maps with $X \in \mathcal{C}$ such that for every $a : A$, $u : MA$ and $f : (MB)^A$

$$\begin{aligned} \tau_A\,(\mathsf{ret}^M_A(a)) : NA &= \mathsf{ret}^N_A(a) \\ \tau_B\,(\mathsf{bind}^M_{A,B}(u, f)) : NB &= \mathsf{bind}^N_{A,B}(\tau_A\,u, \lambda a : A.\,\tau_B\,(f\,a)) \end{aligned}$$

Example 3.8. The monad $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$ of environments in S is

$$\begin{aligned} MX &\triangleq X^S \\ \mathsf{ret}^M_X(x : X) : MX &\triangleq \lambda s : S.\,x \\ \mathsf{bind}^M_{X,Y}(m : MX, f : MY^X) : MY &\triangleq \lambda s : S.\,f\,(m\,s)\,s \end{aligned}$$

The environment monad indexes values by an environment, computations introduced by ret^M ignore the environment, and $\mathsf{bind}^M(m, f)$ applies the same environment to m and to the result of f.

The environment monad has an algebraic operation for the functor $S_{\mathsf{get}}X = X^S$ for reading the environment and a first-order (but not algebraic) operation for the functor $S_{\mathsf{local}}X = S^S \times X$ for performing a computation in a modified environment.

$$\begin{aligned} \mathsf{get}_X\,(f : (MX)^S) : MX &\triangleq \lambda s : S.\,f\,s\,s \\ \mathsf{local}_X(f : S^S, t : MX) : MX &\triangleq \lambda s : S.\,t\,(f\,s) \end{aligned}$$

The more usual operation $\underline{\mathsf{get}}$ for the environment monad is a derived operation which is defined using get.

$$\underline{\mathsf{get}} : ME \triangleq \mathsf{get}_E(\mathsf{ret}^M_E) = \lambda e.\,e$$

Example 3.9. The monad $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$ of side-effects on S is

$$\begin{aligned} MX &\triangleq (X \times S)^S \\ \mathsf{ret}^M_X(x : X) : MX &\triangleq \lambda s : S.\,(x, s) \\ \mathsf{bind}^M_{X,Y}(m : MX, f : MY^X) : MY &\triangleq \lambda s : S.\,\mathsf{let}\,(a, s') = m\,s\,\mathsf{in}\,f\,a\,s' \end{aligned}$$

Intuitively, a computation MX takes an initial state and produces a value of type X and a final state, ret^M does not change the state, and bind^M threads the state.

The side-effect monad has two algebraic operations, one for the functor $S_{\mathsf{get}}X = X^S$ which applies the current state to its argument and the other for the functor $S_{\mathsf{put}}X = S \times X$, which runs a stateful computation in the provided state.

$$\begin{aligned} \mathsf{get}_X\,(k : (MX)^S) : MX &\triangleq \lambda s : S.\,k\,s\,s \\ \mathsf{put}_X\,(s : S, m : MX) : MX &\triangleq \lambda s' : S.\,m\,s \end{aligned}$$

These operations can be used to define the more usual derived operations

$$\underline{\mathsf{get}} : M\,S \mathrel{\hat{=}} \mathsf{get}_S(\mathsf{ret}_S^M) = \lambda s.\,(s,s)$$
$$\underline{\mathsf{put}} : S \to M\,1 \mathrel{\hat{=}} \lambda s : S.\,\mathsf{put}_1(s, \mathsf{ret}_1^M(*)) = \lambda s : S.\,\lambda s' : S.\,(*, s)$$

where $*$ is the sole element of the final object 1.

Example 3.10. The monad $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$ of traces over a monoid $(W, 0, +)$ in **Set** is

$$\begin{aligned} MX &\mathrel{\hat{=}} X \times W \\ \mathsf{ret}_X^M(x : X) : MX &\mathrel{\hat{=}} (x, 0) \\ \mathsf{bind}_{X,Y}^M((x,w) : MX, f : MY^X) : MY &\mathrel{\hat{=}} \mathsf{let}\,(y, w') = f\,x\,\mathsf{in}\,(y, w + w') \end{aligned}$$

A computation MX associates to each value X a trace, ret^M associates a value with the empty trace, and $\mathsf{bind}^M(m, f)$ combines the trace of m and the trace resulting from the application of f to the value of m.

The trace monad has an operation trace that adds an element of the monoid to the trace and it is an algebraic operation for the functor $S_{\mathsf{trace}}X = X \times W$, and an operation flush that erases the trace and it is a first order operation for the functor $S_{\mathsf{flush}}X = X$.

$$\mathsf{trace}_X\,(t : MX, w : W) : MX \mathrel{\hat{=}} \mathsf{let}\,(x, w') = t\,\mathsf{in}\,(x, w' + w)$$
$$\mathsf{flush}_X\,(t : MX) : MX \mathrel{\hat{=}} \mathsf{let}\,(y, w) = t\,\mathsf{in}\,(y, 0)$$

The usual operation $\underline{\mathsf{trace}}$ for the trace monad is a derived operation which is defined using trace.

$$\underline{\mathsf{trace}} : W \to M1 \mathrel{\hat{=}} \lambda w : W.\,\mathsf{trace}_1(w, \mathsf{ret}_1^M(*)) = \lambda w : W.\,(*, w)$$

Example 3.11. The monad $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$ of exceptions in E is

$$\begin{aligned} MX &\mathrel{\hat{=}} X + E \\ \mathsf{ret}_X^M(x : X) : MX &\mathrel{\hat{=}} \mathsf{inl}\,x \\ \mathsf{bind}_{X,Y}^M(m : MX, f : MY^X) : MY &\mathrel{\hat{=}} [f, \mathsf{inr}]\,m \end{aligned}$$

where $[f, g] : A + B \to C$, for $f : A \to C$ and $g : B \to C$, is defined by the universal property of coproducts.

A computation MX is either a pure value in X or an exception in E, ret^M inserts a pure value, and bind^M propagates exceptions.

The exception monad has an algebraic operation for the functor $S_{\mathsf{throw}}X = E$ which throws an exception in E, and a first-order (but not algebraic) operation for the functor $S_{\mathsf{handle}}X = X \times X^E$ that handles exceptions, namely

$$\mathsf{throw}_X\,(e : E) : MX \mathrel{\hat{=}} \mathsf{inr}\,e$$
$$\mathsf{handle}_X : (m : MX,\ h : (MX)^E) : MX \mathrel{\hat{=}} [\mathsf{inl}, h]\,m$$

Example 3.12. The monad $\hat{M} = (M, \mathsf{ret}^M, \mathsf{bind}^M)$ of continuations in R is

$$\begin{aligned} MX &\triangleq R^{(R^X)} \\ \mathsf{ret}^M_X(x : X) : MX &\triangleq \lambda k : R^X.\, k\, x \\ \mathsf{bind}^M_{X,Y}(m : MX, f : MY^X) : MY &\triangleq \lambda k : R^Y.\, m\, (\lambda x : X.\, f\, x\, k) \end{aligned}$$

Intuitively, MX is a computation that given a continuation R^X returns a result in R, ret^M simply runs a continuation, and $\mathsf{bind}^M(m, f)$ runs m with a continuation constructed by running f in the current continuation.

It has two algebraic operations, one for the functor $S_{\mathsf{abort}}X = R$ and the other for the functor $S_{\mathsf{callcc}}X = X^{(R^X)}$, namely

$$\begin{aligned} \mathsf{abort}_X\,(r : R) : MX &\triangleq \lambda k : R^X.\, r \\ \mathsf{callcc}_X(f : (MX)^{(R^{MX})}) : MX &\triangleq \lambda k : R^X.\, f\,(\lambda t : MX.\, t\, k)\, k \end{aligned}$$

The usual call-with-current-continuation $\underline{\mathsf{callcc}}$ is a derived operation which is defined using callcc and abort.

$$\begin{aligned} \underline{\mathsf{callcc}}_{X,Y}\,(f : MX^{(MY^X)}) : MX &\triangleq \mathsf{callcc}\,(\lambda k : R^{MX}.\, f\,(\lambda x : X.\, \mathsf{abort}_Y(k\,(\mathsf{ret}^M x)))) \\ &= \lambda k : R^X.\, f\,(\lambda x : X.\, \lambda k' : R^Y.\, k\, x)\, k \end{aligned} \tag{3.1}$$

Note that R is present in the signatures of the algebraic operations callcc and abort, but this parameter is usually hidden from the user. In this sense, one can think of these operations as being "low-level" operations for the continuation monad, from which higher-level operations are defined, such as $\underline{\mathsf{callcc}}$ in (3.1).

The operations introduced in the examples are summarised in Fig. 3.1. All these operations are first-order, and interestingly, all the operations considered by Liang, Hudak, and Jones (1995) for these monads are definable in terms of these operations.

Monads Induced by Algebraic Theories

Algebraic theories (Manes 1976) are presented by operations and equations. They are a common source of monads with associated operations, since every algebraic theory induces a monad on **Set**.

An *algebraic signature* Σ consists of a set O of operations and a function # assigning to each $o \in O$ its arity $\#o \in$ **Set** (a signature Σ is called *finitary* when each $\#o$ is finite). A signature induces an endofunctor $\Sigma X \triangleq \coprod_{o \in O} X^{\#o}$, which allows to give a concise definition of Σ-*algebra* and Σ-*homomorphism*. Given an endofunctor F (on **Set**) the category F-**Alg** of F-algebras is given by

objects are F-algebras $\underline{A} = (A, \alpha)$, i.e. a set A (the carrier) and a map $FA \xrightarrow{\alpha} A$ (the interpretation of the operations $o \in O$ when $F = \Sigma$)

Monad	Signature	$\hat{\Sigma}$-operations
Environment $MX \triangleq X^S$	$\Sigma^{\mathsf{get}} X \triangleq X^S$ $\Sigma^{\mathsf{local}} X \triangleq S^S \times X$	$\mathsf{get}_X\colon MX^S \to MX$ $\mathsf{local}_X\colon S^S \times MX \to MX$
State $MX \triangleq (X \times S)^S$	$\Sigma^{\mathsf{get}} X \triangleq X^S$ $\Sigma^{\mathsf{put}} X \triangleq S \times X$	$\mathsf{get}_X\colon (MX^S) \to MX$ $\mathsf{put}_X\colon S \times MX \to MX$
Trace over W $MX \triangleq X \times W$	$\Sigma^{\mathsf{trace}} X \triangleq W \times X$ $\Sigma^{\mathsf{flush}} X \triangleq X$	$\mathsf{trace}_X\colon W \times MX \to MX$ $\mathsf{flush}_X\colon MX \to MX$
Exception $MX \triangleq E + X$	$\Sigma^{\mathsf{throw}} X \triangleq E$ $\Sigma^{\mathsf{handle}} X \triangleq X \times (X^E)$	$\mathsf{throw}_X\colon E \to MX$ $\mathsf{handle}_X\colon MX \times (MX^E) \to MX$
Continuation $MX \triangleq R^{(R^X)}$	$\Sigma^{\mathsf{abort}} X \triangleq R$ $\Sigma^{\mathsf{callcc}} X \triangleq X^{(R^X)}$	$\mathsf{abort}_X\colon R \to MX$ $\mathsf{callcc}_X\colon MX^{(R^{MX})} \to MX$

Figure 3.1: First-order operations for the standard monads.

arrows from $\underline{A_1}$ to $\underline{A_2}$ are maps $h : A_1 \longrightarrow A_2$ such that

$$\begin{array}{ccc} FA_1 & \xrightarrow{Fh} & FA_2 \\ {\scriptstyle \alpha_1}\downarrow & & \downarrow{\scriptstyle \alpha_2} \\ A_1 & \xrightarrow[h]{} & A_2 \end{array}$$

Identities and composition are inherited from **Set**.

There is an obvious forgetful functor $U_F : F\textbf{-Alg} \longrightarrow \textbf{Set}$, mapping $\underline{A}$ to A. When Σ is the endofunctor induced by an algebraic signature, U_Σ has a left adjoint F_Σ ($F_\Sigma X$ is called the free Σ-algebra over X, and an element t in its carrier is called a Σ-term with free variables in X). The monad induced by the adjunction $F_\Sigma \dashv U_\Sigma$ is the free monad Σ^* over Σ (see Example 2.17), and Σ**-Alg** is isomorphic to the category $\textbf{Set}^{\Sigma^*}$ of Eilenberg-Moore algebras for the monad Σ^*.

An *algebraic theory* $T = (\Sigma, Eq)$ consists of an algebraic signature Σ and a set Eq of equations between Σ-terms (with free variables in some set X). The theory T induces a full subcategory T**-Alg** of Σ**-Alg**, whose objects are the Σ-algebras *satisfying* all the equations in Eq. Also in this case there is a forgetful functor $U_T : T\textbf{-Alg} \longrightarrow \textbf{Set}$ (the restriction of U_Σ to T**-Alg**), which has a left adjoint F_T. The adjunction $F_T \dashv U_T$ induces a monad M_T on **Set**, and the category T**-Alg** is isomorphic to the category $\textbf{Set}^{M_T}$ of Eilenberg-Moore algebras for the monad M_T (see Mac Lane 1971).

All monads given in this section, except that in Example 3.12, are induced by algebraic theories. Moreover, all monads for *collection types* (such as lists, bags, sets) arise from *balanced*

finitary algebraic theories (see Manes 1998). An algebraic theory is balanced when for each equation, the same set of variables occurs on both terms of the equation.

In the following examples, we write equations in the style of Universal Algebra and write $(t_i \mid i \in S)$ for an S-indexed set.

- The monad of Example 3.8 $MX = X^S$ corresponds to T_{env} given by $O = \{\mathsf{get}\}$, $\#\mathsf{get} = S$ and the equation

$$\mathsf{get}\,(\mathsf{get}\,(t_{i,j} \mid j \in S) \mid i \in S) = \mathsf{get}\,(t_{i,i} \mid i \in S)$$

- The monad of Example 3.9 $MX = (X \times S)^S$ corresponds to T_{state} given by T_{env} extended with $O = \{\mathsf{put}_s \mid s \in S\}$, $\#\mathsf{put}_s = 1$ and equations

$$\begin{aligned} \mathsf{get}(t_i \mid i \in S) &= \mathsf{get}(\mathsf{put}_i(t_i) \mid i \in S) \\ \mathsf{put}_i(\mathsf{get}(t_j \mid j \in S)) &= \mathsf{put}_i(t_i) \quad \text{with } i \in S \\ \mathsf{put}_i(\mathsf{put}_j(t)) &= \mathsf{put}_j(t) \quad \text{with } i, j \in S \end{aligned}$$

- The monad of Example 3.10 $MX = X \times W$, where $(W, 0, +)$ is a monoid, corresponds to T_W given by $O = \{\mathsf{trace}_w \mid w \in W\}$, $\#\mathsf{trace}_w = 1$ and equations

$$\begin{aligned} \mathsf{trace}_0(t) &= t \\ \mathsf{trace}_i(\mathsf{trace}_j(t)) &= \mathsf{trace}_{i+j}(t) \quad \text{with } i, j \in W \end{aligned}$$

- The monad of Example 3.11 $MX = X + E$ corresponds to T_{exc} given by

  $O = \{\mathsf{throw}_e \mid e \in E\}$, $\#\mathsf{throw}_e = 0$ and no equations

- The *list* monad $MX = X^*$ corresponds to T_{list} given by $O = \{\mathsf{nil}, \mathsf{append}\}$, $\#\mathsf{nil} = 0$, $\#\mathsf{append} = 2$ and equations

$$\mathsf{append}(\mathsf{nil}, t) = t = \mathsf{append}(t, \mathsf{nil})$$

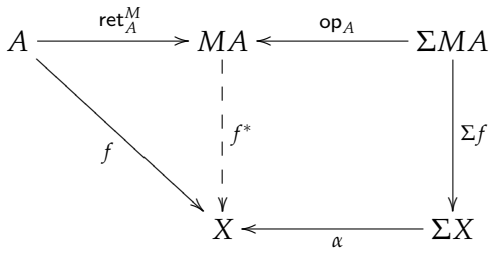
$$\mathsf{append}(\mathsf{append}(t_1, t_2), t_3) = \mathsf{append}(t_1, \mathsf{append}(t_2, t_3))$$

- The (finite) *set* monad corresponds to T_{list} extended with the equations

$$\begin{aligned} \mathsf{append}(t_1, t_2) &= \mathsf{append}(t_2, t_1) \\ \mathsf{append}(t, t) &= t \end{aligned}$$

The monad $\hat{M}$ induced by an algebraic theory $T = (\Sigma, Eq)$ has an associated algebraic operation $\mathsf{op}_X : \Sigma(MX) \longrightarrow MX$ of signature Σ, where op_X is the Σ-algebra structure on MX. When $\hat{M}$ is the free monad Σ^*, i.e. $T = (\Sigma, \emptyset)$, one can associate to $\hat{M}$ two other operations

- $\mathsf{elim}_X : X^{\Sigma X} \times X^A \longrightarrow X^{MA}$ captures *initiality* of MA among the Σ-algebras over A, namely $\mathsf{elim}_X(\alpha, f)$ is the unique Σ-homomorphism f^* from $\Sigma(MA) \xrightarrow{\mathsf{op}_A} MA$ (the free algebra over A) to $\Sigma X \xrightarrow{\alpha} X$ such that $f^* \circ \mathsf{ret}^M_A = f$. The following diagram illustrates how elim captures initiality:



  The operation elim generalizes $\mathsf{bind}^M_{A,X}$ and, in general, cannot be presented as an H-operation.

- $\underline{\mathsf{case}}_X : MA \times X^A \times X^{\Sigma(MA)} \longrightarrow X$ does case analysis on MA, which is isomorphic to $A + \Sigma(MA)$. The instance of $\underline{\mathsf{case}}$ obtained by replacing X with MX, i.e. $\mathsf{case}_X : MA \times (MX)^A \times (MX)^{\Sigma(MA)} \longrightarrow MX$, can be presented as an H-operation for $H\hat{N}X \mathrel{\hat{=}} NA \times (NX)^A \times (NX)^{\Sigma(MA)}$, provided the M in contravariant position is fixed.

3.3 Examples of Lifted Algebraic Operations

The following examples show the operations obtained by applying the algebraic lifting to the algebraic operations get and callcc.

Example 3.13. Let $\hat{M}$ be the side-effect monad of Example 3.9, and consider the monad $\hat{N}$—which will be shown in the next chapter to be the result of applying the exceptions monad transformer to the side-effects monad—given by $NX = S \to ((X + E) \times S)$ and the monad morphism $\xi : M \xrightarrow{\bullet} N$ defined as

$$\xi_X(t : MX) \mathrel{\hat{=}} \mathsf{bind}^M_{X,X+E}(t, \lambda x : X.\, \mathsf{ret}^M_X(\mathsf{inl}\ x)).$$

The algebraic lifting of the algebraic operation get yields the operation

$$\mathsf{get}^N_X(k : S \to NX) : NX \mathrel{\hat{=}} \lambda s : S.\, k\, s\, s.$$

Example 3.14. Let $\hat{M}$ be the continuation monad of Example 3.12, and consider the monad $\hat{N}$—which will be shown in the next chapter to be the result of applying the side-effect monad transformer to the continuation monad—given by $NX = (R^{(R^{X \times S})})^S$ and the monad morphism $\xi : M \xrightarrow{\bullet} N$ defined as

$$\xi_X(t : MX) \mathrel{\hat{=}} \lambda s : S.\, \mathsf{bind}^M_{X,X\times S}(t, \lambda x : X.\, (x, s)).$$

The operation obtained by the algebraic lifting of callcc simplifies to:

$$\mathsf{callcc}^N\,(f : NX^{(R^{NX})}) : NX = \lambda s : S.\,\lambda k : R^{X\times S}.\,f\,(\lambda n : NX.\,n\,s\,k)\,s\,k.$$

We can define a lifted version of $\underline{\mathsf{callcc}}$ using equation 3.1, callcc^N, and abort^N (the unique algebraic lifting of abort) and obtain:

$$\begin{aligned}\underline{\mathsf{callcc}}^N\,(f : NX^{(NY^X)}) : NX &\triangleq \mathsf{callcc}^N\,(\lambda k : R^{NX}.\,f\,(\lambda x : X.\,\mathsf{abort}^N_Y(k\,(\mathsf{ret}^N x))))\\ &= \lambda s : S.\,\lambda k : R^{X\times S}.\,f\,(\lambda x : X.\,\lambda s' : S.\,\lambda k' : R^{Y\times S}.\,k\,x\,s)\,s\,k.\end{aligned}$$

The author has used the algebraic lifting of callcc to verify the ad-hoc liftings of callcc in Haskell's monad transformer library (mtl). This verification has revealed that the uniform lifting above coincided with all of the library's liftings, except for one: the library's lifting of callcc to the monad $\hat{N}$ defined above is not consistent with the rest of the liftings. The ad-hoc lifting of $\underline{\mathsf{callcc}}$ in mtl is:

$$\underline{\mathsf{callcc{-}mtl}}^N\,(f : NX^{(NY^X)}) : NX = \lambda s : S.\,\lambda k : R^{X\times S}.\,f\,(\lambda x : X.\,\lambda s' : S.\,\lambda k' : R^{Y\times S}.\,k\,x\,s')\,s\,k.$$

The difference is that the ad-hoc lifted operation preserves changes in the state produced during the construction of the new continuation even when the current continuation is used. However, all the other liftings of $\underline{\mathsf{callcc}}$ in the library do not preserve produced effects when using the current continuation. Consequently, that particular lifting in the mtl is not coherent with the other liftings of callcc.

3.4 Summary

We have defined a general notion of operation on a monoid, and two refinements: first-order operations and algebraic operations. Moreover we have shown that all algebraic operations can be lifted along any monoid morphism.

We have shown many examples of monads and some of their associated operations. Although all of the operations in Figure 3.1 are algebraic or first-order, they are enough to define all the operations considered by Liang, Hudak, and Jones (1995). Comparing these operations with the operations provided by Haskell's mtl (*Monad Transformer Library*), the only operation not covered is the operation of the trace monad $\mathsf{listen} :: MX \to M(X \times W)$. However, listen can be defined from the following H-operation:

$$\mathsf{collect}_{A,X}(t : MA, f : MX^{(A\times W)}) \triangleq \mathsf{let}\,(y, w) = t\,\mathsf{in}\,(f\,t, w)$$

Remarkably, as shown in Example 3.12, callcc is an algebraic operation despite not being algebraic in the sense of Plotkin and Power (2001*b*) and hence, not tractable in that approach. With the generalization made here, callcc is not only tractable, but also well-behaved. As a consequence callcc easily lifts along any monad morphism.

Many of the monads presented in this chapter arise from algebraic theories (Manes 1976). Interestingly, these monads can arise from computationally natural operations and equations, as shown by Plotkin and Power (2002). Lawvere theories (Hyland and Power 2007; Lawvere 1963) provide another way of modelling algebraic theories categorically. A more general way to get monads, not pursued here, is through the *equational systems* of Fiore and Hur (2009).

As shown at the end of Section 3.2, some operations such as elim and case are not, in general, H-operations (the most general class of operations in this thesis). The operation elim is related to the try construct in Plotkin and Pretnar (2009). These operations—which are not H-operations—indicate an obvious direction for further extension of the framework.

Chapter 4

Monoid Transformers and Operation Lifting

In this chapter, the notion of a transformer is introduced, and a theory of monoid transformers is developed. In particular, a hierarchy of monoid transformers in the setting of a monoidal category $\hat{\mathcal{E}}$ is defined and several lifting theorems are proved. The lifting theorems show how certain classes of operations can be lifted through particular classes of monoid transformers. For a given operation and monoid transformer, several lifting results may be applicable, but it is shown that, when more than one lifting result is applicable, then the liftings coincide. The chapter also provides various examples of monoid transformers motivated by the incremental approach to monadic semantics and examples of operations lifted through these transformers.

4.1 Monoid Transformers

Given a category $\mathcal{A}$, a *transformer* on $\mathcal{A}$ is a 2-cell

$$\mathcal{A}' \;\underset{T}{\overset{\mathsf{In}}{\Downarrow \mathsf{lift}^T}}\; \mathcal{A} \quad \text{where}$$

- $\mathcal{A}'$ is a sub-category of $\mathcal{A}$, such as $\mathcal{A}$ itself or the *discrete category* $|\mathcal{A}|$ with the same objects as $\mathcal{A}$,
- $\mathsf{In} : \mathcal{A}' \longrightarrow \mathcal{A}$ is the *inclusion functor*,
- $T : \mathcal{A}' \longrightarrow \mathcal{A}$ is a functor, and
- $\mathsf{lift}^T : \mathsf{In} \xrightarrow{\bullet} T$ is a natural transformation, therefore $\mathsf{lift}^T_a : a \longrightarrow Ta$ is a map in $\mathcal{A}$ for any $a \in \mathcal{A}'$.

Monoid transformers are simply transformers on $\mathsf{Mon}(\hat{\mathcal{E}})$. In the following, a new hierarchy of monad transformers is introduced (Jaskelioff and Moggi 2009). The minimum requirement on a monoid transformer T is when the subcategory is $|\mathsf{Mon}(\hat{\mathcal{E}})|$ and T simply maps a monoid $\hat{M} \in \mathsf{Mon}(\hat{\mathcal{E}})$ to a monoid $T\hat{M}$ (and provides a natural transformation $\mathsf{In} \longrightarrow T$). The maximum requirement in the hierarchy is a monoid transformer T induced by a monoidal endofunctor $\hat{T}$ on $\hat{\mathcal{E}}$ and a monoidal natural transformation $\hat{\mathsf{Id}} \to \hat{T}$.

Definition 4.1 (Monoid Transformers)**.**

1. A *monoid transformer* is a pair (T, lift^T) such that

$$|\mathsf{Mon}(\hat{\mathcal{E}})| \underset{T}{\overset{\mathsf{In}}{\Downarrow \mathsf{lift}^T}} \mathsf{Mon}(\hat{\mathcal{E}})$$

2. A *covariant* monoid transformer is a pair (T, lift^T) such that

$$\mathsf{Mon}(\hat{\mathcal{E}}) \underset{T}{\overset{\mathsf{Id}}{\Downarrow \mathsf{lift}^T}} \mathsf{Mon}(\hat{\mathcal{E}})$$

3. A *functorial* monoid transformer is a covariant monoid transformer (T, lift^T) with an underlying transformer on $\mathcal{E}$, also denoted (T, lift^T), i.e.

$$\begin{array}{ccc} \mathsf{Mon}(\hat{\mathcal{E}}) & \underset{T}{\overset{\mathsf{Id}}{\Downarrow \mathsf{lift}^T}} & \mathsf{Mon}(\hat{\mathcal{E}}) \\ U \downarrow & & \downarrow U \\ \mathcal{E} & \underset{T}{\overset{\mathsf{Id}}{\Downarrow \mathsf{lift}^T}} & \mathcal{E} \end{array}$$

   In particular $U(\mathsf{lift}^T_{\hat{M}}) = \mathsf{lift}^T_{U\hat{M}} = \mathsf{lift}^T_M$.

4. A *monoidal* monoid transformer is a functorial monoid transformer (T, lift^T) induced by a transformer $\hat{\mathcal{E}} \underset{\hat{T}}{\overset{\hat{\mathsf{Id}}}{\Downarrow \mathsf{lift}^T}} \hat{\mathcal{E}}$ with $\hat{T}$ a monoidal endofunctor and lift^T a monoidal natural transformation (see Theorem 2.7).

From the definitions and Theorem 2.7, we get that

$$\text{monoidal} \subset \text{functorial} \subset \text{covariant} \subset \text{transformer.}$$

These are proper inclusions, as shown in Example 4.8.

4.2 Examples of Transformers

This section presents examples of strong monad transformers on a cartesian closed category $\mathcal{C}$, i.e. monoid transformers on the strict monoidal category of strong endofunctors on $\mathcal{C}$. Some examples require additional assumptions on $\mathcal{C}$ besides cartesian closure. Although cartesian closure is assumed for the examples in this section, the definition of strong endofunctors can be given assuming only a cartesian category. There are equivalent ways of defining strong endofunctors on a cartesian closed category $\mathcal{C}$. As already done for strong monads (see Definition 3.6), we borrow the definition adopted in Haskell, and freely use simply typed lambda-calculus as *internal language* to denote objects and maps in $\mathcal{C}$. Again, we freely use syntactic sugar typical of typed lambda-calculi such as let constructs and pattern matching on products.

Definition 4.2 (Strong Endofunctor)**.** A *strong endofunctor* on a cartesian closed category $\mathcal{C}$ is a pair $F = (F, \mathsf{map}^F)$ consisting of

- a map $F : |\mathcal{C}| \longrightarrow |\mathcal{C}|$ on the objects of $\mathcal{C}$
- a family $\mathsf{map}^F_{X,Y} : Y^X \times FX \longrightarrow FY$ of maps with $X, Y \in \mathcal{C}$

such that for every $u : FA$, $f : B^A$ and $g : C^B$ the following equations hold.

$$\mathsf{map}^F_{A,A}(\mathsf{id}_A,\, u) : FA = u$$
$$\mathsf{map}^F_{A,C}(g \circ f,\, u) : FC = \mathsf{map}^F_{B,C}(g,\, \mathsf{map}^F_{A,B}(f,\, u))$$

A *strong natural transformation* $\tau : F \longrightarrow G$ is a family $\tau_X : FX \longrightarrow GX$ of maps with $X \in \mathcal{C}$ such that for every $u : FA$ and $f : B^A$ the following equation holds.

$$\tau_B(\mathsf{map}^F_{A,B}(f,\, u)) : GB = \mathsf{map}^G_{A,B}(f,\, \tau_A(u))$$

Example 4.3. The transformer (T, lift^T) for adding *environments* in $S \in \mathcal{C}$ is

- T maps a strong monad $\hat{M}$ to the strong monad $\hat{N}$ given by

$$NX \mathrel{\hat{=}} (MX)^S$$
$$\mathsf{ret}^N_X(x) : NX \mathrel{\hat{=}} \lambda s : S.\, \mathsf{ret}^M_X(x)$$
$$\mathsf{bind}^N_{X,Y}(c, f) : NY \mathrel{\hat{=}} \lambda s : S.\, \mathsf{bind}^M_{X,Y}(c\, s, \lambda x : X.\, f\, x\, s)$$

- lift^T maps a strong monad $\hat{M}$ to $\tau : \hat{M} \longrightarrow T\hat{M}$ given by

$$\tau_X(c : MX) : T\hat{M}X \mathrel{\hat{=}} \lambda s : S.\, c$$

This transformer is monoidal. More precisely, it is induced by the following monoidal functor $\hat{T} = (T, \phi_I, \phi)$ and monoidal natural transformation lift^T

- T maps a strong functor F to the strong functor G given by

$$GX \triangleq (FX)^S$$
$$\mathsf{map}^G_{X,Y}(f,\, u) : GY \triangleq \lambda s : S.\, \mathsf{map}^F_{X,Y}(f,\, u\, s)$$

  and maps $\tau : F_1 \xrightarrow{\bullet} F_2$ to $T\tau : TF_1 \xrightarrow{\bullet} TF_2$ given by

$$(T\tau)_X(u) : TF_2X \triangleq \lambda s : S.\, \tau_X(u\, s)$$

- $\phi_I : \mathsf{Id} \xrightarrow{\bullet} T(\mathsf{Id})$ and $\phi_{F_2,F_1} : TF_2 \circ TF_1 \xrightarrow{\bullet} T(F_2 \circ F_1)$ are

$$\phi_{I,X}(x : X) : T(\mathsf{Id})X \triangleq \lambda s : S.\, x$$
$$\phi_{F_2,F_1,X}(u : (F_2((F_1X)^S))^S) : T(F_1 \circ F_2)X \triangleq \lambda s : S.\, \mathsf{map}^{F_2}_{(F_1X)^S,F_1X}(\lambda f : (F_1X)^S.\, f\, s,\, u\, s)$$

- $\mathsf{lift}^T_F : F \xrightarrow{\bullet} TF$ is $\mathsf{lift}^T_{F,X}(u : FX) : TFX \triangleq \lambda s : S.\, u$

Example 4.4. The transformer (T, lift^T) for adding *side-effects* on $S \in \mathcal{C}$ is

- T maps a strong monad $\hat{M}$ to the strong monad $\hat{N}$ given by

$$NX \triangleq (M(X \times S))^S$$
$$\mathsf{ret}^N_X(x) : NX \triangleq \lambda s : S.\, \mathsf{ret}^M_{X\times S}(x, s)$$
$$\mathsf{bind}^N_{X,Y}(c, f) : NY \triangleq \lambda s : S.\, \mathsf{bind}^M_{X\times S,Y\times S}(c\, s, \lambda(x : X, s' : S).\, f\, x\, s')$$

- lift^T maps a strong monad $\hat{M}$ to $\tau : \hat{M} \longrightarrow T\hat{M}$ given by

$$\tau_X(c : MX) : T\hat{M}X \triangleq \lambda s : S.\, \mathsf{bind}^M_{X,X\times S}(c, \lambda x : X.\, \mathsf{ret}^M_{X\times S}(x, s))$$

This transformer is monoidal. More precisely, it is induced by the following monoidal functor $\hat{T}$ and monoidal natural transformation lift^T

- T maps a strong functor F to the strong functor G given by

$$GX \triangleq (F(X \times S))^S$$
$$\mathsf{map}^G_{X,Y}(f,\, u) : GY \triangleq \lambda s : S.\, \mathsf{map}^F_{X\times S,Y\times S}(\lambda(x : X, s' : S).\, (f\, x, s'),\, u\, s)$$

  and maps $\tau : F_1 \xrightarrow{\bullet} F_2$ to $T\tau : TF_1 \xrightarrow{\bullet} TF_2$ given by

$$(T\tau)_X(u) : TF_2X \triangleq \lambda s : S.\, \tau_{X\times S}(u\, s)$$

- $\phi_I : \mathsf{Id} \xrightarrow{\bullet} T(\mathsf{Id})$ and $\phi_{F_2,F_1} : TF_2 \circ TF_1 \xrightarrow{\bullet} T(F_2 \circ F_1)$ are

$$\begin{aligned}\phi_{I,X}(x : X) : T(\mathsf{Id})X &\triangleq \lambda s : S.\, (x, s)\\ \phi_{F_2,F_1,X}(u : (F_2(F_1(X \times S)^S \times S))^S) : T(F_2 \circ TF_1)X &\triangleq \\ &\lambda s : S.\, \mathsf{map}^{F_2}_{F_1(X\times S)^S\times S,F_1(X\times S)}(\lambda(f : F_1(X \times S)^S, s' : S).\, f\, s',\, u\, s)\end{aligned}$$

- $\mathsf{lift}^T_F : F \xrightarrow{\bullet} TF$ is $\mathsf{lift}^T_{F,X}(u : FX) : TFX \mathrel{\hat{=}} \lambda s : S.\, \mathsf{map}^F_{X,X\times S}(\lambda x : X.\, (x, s),\, u)$

Example 4.5. The transformer (T, lift^T) for adding *traces* over a monoid $(W, 0, +)$ in $\mathcal{C}$ is

- T maps a strong monad $\hat{M}$ to the strong monad $\hat{N}$ given by

$$\begin{aligned} NX &\mathrel{\hat{=}} M(X \times W) \\ \mathsf{ret}^N_X(x) : NX &\mathrel{\hat{=}} \mathsf{ret}^M_{X\times W}(x, 0) \\ \mathsf{bind}^N_{X,Y}(c, f) : NY &\mathrel{\hat{=}} \mathsf{bind}^M(c, \lambda(x : X, w : W).\, \mathsf{bind}^M(f\, x, \lambda(y : Y, w' : W).\, \mathsf{ret}^M(y, w + w'))) \end{aligned}$$

- lift^T maps a strong monad $\hat{M}$ to $\tau : \hat{M} \longrightarrow T\hat{M}$ given by

$$\tau_X(c : MX) : T\hat{M}X \mathrel{\hat{=}} \mathsf{bind}^M_{X,X\times W}(c, \lambda x : X.\, \mathsf{ret}^M_{X\times W}(x, 0))$$

This transformer is also monoidal. More precisely, it is induced by the following monoidal functor $\hat{T}$ and monoidal natural transformation lift^T

- T maps a strong functor F to the strong functor G given by

$$\begin{aligned} GX &\mathrel{\hat{=}} F(X \times W) \\ \mathsf{map}^G_{X,Y}(f,\, u) : GY &\mathrel{\hat{=}} \mathsf{map}^F_{X\times W, Y\times W}(\lambda(x : X, w : W).\, (f\, x, w),\, u) \end{aligned}$$

  and maps $\tau : F_1 \xrightarrow{\bullet} F_2$ to $T\tau : TF_1 \xrightarrow{\bullet} TF_2$ given by

$$(T\tau)_X(u) : TF_2X \mathrel{\hat{=}} \tau_{X\times W}(u)$$

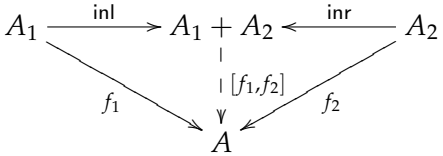
- $\phi_I : \mathsf{Id} \xrightarrow{\bullet} T(\mathsf{Id})$ and $\phi_{F_2,F_1} : TF_2 \circ TF_1 \xrightarrow{\bullet} T(F_2 \circ F_1)$ are

$$\begin{aligned} \phi_{I,X}(x : X) : T(\mathsf{Id})X &\mathrel{\hat{=}} (x, 0) \\ \phi_{F_2,F_1,X}(u : F_2(F_1(X \times W) \times W)) : T(F_2 \circ F_1)X &\mathrel{\hat{=}} \\ &\mathsf{map}^{F_2}(\lambda(f, w).\, \mathsf{map}^{F_1}(\lambda(x, w').\, (x, w' + w),\, f),\, u) \end{aligned}$$

  where some type information has been omitted for readability.

- $\mathsf{lift}^T_F : F \xrightarrow{\bullet} TF$ is $\mathsf{lift}^T_{F,X}(u : FX) : TFX \mathrel{\hat{=}} \mathsf{map}^F_{X,X\times W}(\lambda x : X.\, (x, 0),\, u)$

Example 4.6. In this example we need additional assumptions on $\mathcal{C}$, namely

- existence of binary sums 

  (we write $f_1 + f_2$ for the action of $+$ on maps), and

- existence of initial algebras $\alpha_F : F(\mu X.\, FX) \longrightarrow \mu X.\, FX$ for every strong endofunctor $\hat{F}$ (for simplicity, we assume that α_F is the identity map).

To satisfy these assumptions one could take as $\mathcal{C}$ the cartesian closed category $\mathcal{P}_A$ of partial equivalence relations, and instead of $\mathsf{Endo}(\mathcal{P}_A)_s$ use the more restricted category $\mathsf{Endo}(\mathcal{P}_A)_r$ of realisable endofunctors and realisable natural transformations (as done in Example 2.10), take $\mathcal{C}$ to be a locally finitely presentable category such as **Set**, and instead of $\mathsf{Endo}(\mathcal{C})$ use the full subcategory of finitary endofunctors $\mathsf{Endo}(\mathcal{C})_f$ (see Example 2.9), or use the subcategory of containers and container morphisms (Abbott, Altenkirch, and Ghani 2003). Given an endofunctor S, the transformer (T, lift^T) for adding *S-steps* is

- T maps a monad $\hat{M}$ to the monad $\hat{N}$ given by

$$\begin{aligned} NX &\triangleq \mu X'.\, M(X + SX') \\ \mathsf{ret}^N_X(x) : NX &\triangleq \mathsf{ret}^M_{X+S(NX)}(\mathsf{inl}\, x) \\ \mathsf{step}_X : S(NX) &\longrightarrow NX \\ \mathsf{step}_X(u) &\triangleq \mathsf{ret}^M_{X+S(NX)}(\mathsf{inr}\, u) \\ \mathsf{bind}^N_{X,Y}(c, f) : NY &\triangleq h\, c \end{aligned}$$

  where $NX \xrightarrow{h} NY$ is the unique $M(X + S-)$-algebra morphism from the initial algebra to $\beta : M(X + S(NY)) \longrightarrow NY$ given by

$$\beta(c) \triangleq \mathsf{bind}^M_{X+S(NY),Y+S(NY)}(c, [f, \mathsf{step}_Y])$$

- lift^T maps a monad $\hat{M}$ to $\tau : \hat{M} \longrightarrow \hat{N} = T\hat{M}$ given by

$$\tau_X(c : MX) : NX \triangleq \mathsf{bind}^M_{X,X+S(NX)}(c, \mathsf{ret}^N_X)$$

This transformer is functorial, where the underlying endofunctor transformer (T, lift^T) is

- T maps a functor F to the functor G given by

$$\begin{aligned} GX &\triangleq \mu X'.\, F(X + SX') \\ \mathsf{map}^G_{X,Y}(f,\, u) : GY &\triangleq h\, u \end{aligned}$$

  where $GX \xrightarrow{h} GY$ is the unique $F(X + S-)$-algebra morphism from the initial algebra to $\beta : F(X + S(GY)) \longrightarrow GY$ given by

$$\beta(u) \triangleq \mathsf{map}^F_{X+S(GY),Y+S(GY)}(f + \mathsf{id}_{S(GY)},\, u)$$

  and maps $\tau : F_1 \xrightarrow{\bullet} F_2$ to $T\tau : TF_1 = G_1 \xrightarrow{\bullet} G_2 = TF_2$ given by

$$(T\tau)_X(u) : G_2X \triangleq h\, u$$

where $G_1X \xrightarrow{h} G_2X$ is the unique $F_1(X + S-)$-algebra morphism from the initial algebra to $\beta : F_1(X + S(G_2X)) \longrightarrow G_2X$ given by

$$\beta(u) \triangleq \tau_{X+S(G_2X)}(u)$$

- lift^T maps a realisable endofunctor F to $\tau : F \longrightarrow G = TF$ given by

$$\tau_X(u : FX) : GX \triangleq \mathsf{map}^F_{X,X+S(GX)}(\mathsf{inl},\, u)$$

As shown in Example 4.8, this transformer may fail to be monoidal.

The transformer for adding $\hat{S}$-steps is sometimes referred to as the *free-monad* transformer. The transformer $T\hat{M}$ corresponds to coproduct of the free monad on $\hat{S}$ and $\hat{M}$ in the category of monads over $\mathcal{C}$, as shown by Hyland, Plotkin, and Power (2006).

Example 4.7. We define the list transformer, which needs additional assumptions, like those identified in Example 4.6. The *list* transformer (T, lift^T) is

- T maps a monad $\hat{M}$ to the monad $\hat{N}$ given by

$$\begin{aligned}
NX &\triangleq \mu Y.\, M(1 + X \times Y) \\
\mathsf{nil}_X &: NX \\
\mathsf{nil}_X &\triangleq \mathsf{ret}^M_{1+X\times NX}(\mathsf{inl}\, *) \\
\mathsf{cons}_X &: X \times NX \longrightarrow NX \\
\mathsf{cons}_X(x, l) &\triangleq \mathsf{ret}^M_{1+X\times NX}(\mathsf{inr}(x, l)) \\
\mathsf{ret}^N_X(x) : NX &\triangleq \mathsf{cons}_X(x, \mathsf{nil}_X) \\
\mathsf{bind}^N_{X,Y}(c, f) : NY &\triangleq h\, c
\end{aligned}$$

where $NX \xrightarrow{h} NY$ is the unique $M(1 + X \times -)$-algebra morphism from the initial algebra to $\beta : M(1 + X \times NY) \longrightarrow NY$ given by

$$\beta(c) \triangleq \mathsf{bind}^M_{1+X\times NY,1+Y\times NY}(c, [\mathsf{nil}_Y, \lambda(x, l).\, \mathsf{app}_Y((f\, x), l)])$$

with $NX \xrightarrow{\Lambda \mathsf{app}_X} (NX)^{NX}$ the unique $M(1 + X \times -)$-algebra from the initial algebra to $\Lambda\beta : M(1 + X \times (NX)^{NX}) \longrightarrow (NX)^{NX}$ given by

$$\beta(c, l) \triangleq \mathsf{bind}^M_{1+X\times(NX)^{NX},1+X\times NX}(c, [\mathsf{nil}_X, \lambda(x, f).\, \mathsf{cons}_X(x, f\, l)])$$

To prove that ret^N and bind^N satisfy the equations in Definition 3.6, one can use the following properties of nil_X, cons_X and app_X

$$\begin{aligned}
\mathsf{app}_X(\mathsf{nil}_X, l) &= l = \mathsf{app}_X(l, \mathsf{nil}_X) \\
\mathsf{app}_X(\mathsf{cons}_X(x, l_1), l_2) &= \mathsf{cons}_X(x, \mathsf{app}_X(l_1, l_2)) \\
\mathsf{app}_X(\mathsf{app}_X(l_1, l_2), l_3) &= \mathsf{app}_X(l_1, \mathsf{app}_X(l_2, l_3))
\end{aligned}$$

- lift^T maps a monad $\hat{M}$ to $\tau : \hat{M} \longrightarrow \hat{N} = T\hat{M}$ given by

$$\tau_X(c : MX) : T\hat{M}X \mathrel{\hat{=}} \mathsf{bind}^M_{X,1+X\times NX}(c, \mathsf{ret}^N_X)$$

This transformer is functorial, where the underlying endofunctor transformer (T, lift^T) is

- T maps a functor F to the functor G given by

$$GX \mathrel{\hat{=}} \mu X'.\, F(1 + X \times X')$$
$$\mathsf{map}^G_{X,Y}(f,\, u) \mathrel{\hat{=}} h\, u$$

where $GX \xrightarrow{h} GY$ is the unique $F(1 + X \times -)$-algebra morphism from the initial algebra to $\beta : F(1 + X \times GY) \longrightarrow GY$ given by

$$\beta(u) \mathrel{\hat{=}} \mathsf{map}^F_{1+X\times GY,1+Y\times GY}(\mathsf{id}_1 + (f \times \mathsf{id}_{GY}),\, u)$$

and maps $\tau : F_1 \xrightarrow{\bullet} F_2$ to $T\tau : TF_1 = G_1 \xrightarrow{\bullet} G_2 = TF_2$ given by

$$(T\tau)_X(u) : G_2X \mathrel{\hat{=}} h\, u$$

where $G_1X \xrightarrow{h} G_2X$ is the unique $F_1(1 + X \times -)$-algebra morphism from the initial algebra to $\beta : F_1(1 + X \times G_2X) \longrightarrow G_2X$ given by

$$\beta(u) \mathrel{\hat{=}} \tau_{1+X\times G_2X}(u)$$

- lift^T maps an endofunctor F to $\tau : F \longrightarrow G = TF$ given by

$$\tau_X(u : FX) \mathrel{\hat{=}} \mathsf{map}^F_{X,1+X\times GX}(\mathsf{inr}',\, u)$$

where $\mathsf{inr}' : X \longrightarrow 1 + X \times GX$ is given by

$$\mathsf{inr}'(x) \mathrel{\hat{=}} \mathsf{inr}(x, \mathsf{map}^F_{X,1+X\times GX}(\lambda - .\, \mathsf{inl}\, *,\, u))$$

Example 4.8. We give three (strong) monad transformers on **Set**, which show that the inclusions of the monoid transformer hierarchy are proper. When convenient, we use the fact that every endofunctor/monad on **Set** is strong (see Section 3.2).

1. The transformer (T, lift^T) for adding **continuations** is defined as follows, T maps a strong monad $\hat{M}$ to the strong monad $\hat{N}$ of continuations in MR (see Example 3.12)

$$NX \mathrel{\hat{=}} (MR)^{((MR)^X)}$$
$$\mathsf{ret}^N_X(x) : NX \mathrel{\hat{=}} \lambda k : (MR)^X.\, k\, x$$
$$\mathsf{bind}^N_{X,Y}(c, f) : NY \mathrel{\hat{=}} \lambda k : (MR)^Y.\, c\, (\lambda x : X.\, f\, x\, k)$$

and lift^T maps $\hat{M}$ to the morphism $\tau : \hat{M} \longrightarrow T\hat{M}$ given by

$$\tau_X(c : MX) \mathrel{\hat{=}} \lambda k : (MR)^X.\, \mathsf{bind}^M_{X,R}(c, k)$$

This transformer **is not covariant**, because M is used in contravariant position in NX.

2. Given a strong monad $\hat{M}$, we say that a computation $c : MX$ is **idempotent** when $c = c;c$ where $c_1;c_2 \triangleq \mathsf{bind}^M_{X,X}(c_1, \lambda x : X.\, c_2)$.

   The transformer (T, lift^T) **making computations idempotent** is defined as follows, T maps a strong monad $\hat{M}$ to the smallest quotient monad (Manes 1998) generated by the family of relations

   $$R_X \triangleq \{(c, c;c) \mid c \in MX\}$$

   and $\mathsf{lift}^T_{\hat{M}}$ is the epimorphism from $\hat{M}$ to the quotient monad.

   This transformer is covariant, because $\tau_X(c;c) = \tau_X(c);\tau_X(c) : NX$ for any strong monad morphism $\tau : \hat{M} \longrightarrow \hat{N}$ and $c : MX$, but it **is not functorial**. In fact, there are two trace monads $\hat{M}$ and $\hat{N}$ (see Example 3.10) with the same underlying endofunctor $F(-) \triangleq - \times \mathsf{bool}$, with bool the set of booleans, such that $T\hat{M} = \hat{M}$ and $T\hat{N} = \hat{\mathsf{Id}}$:

   - $\hat{M}$ is the strong monad induced by the monoid $(\mathsf{bool}, \mathsf{false}, \mathsf{or})$ in **Set**. Since this monoid is idempotent, all computations in MX are already idempotent, therefore $T\hat{M} = \hat{M}$.
   - $\hat{N}$ is the strong monad induced by the monoid $(\mathsf{bool}, \mathsf{false}, \mathsf{xor})$ in **Set**. Since $\mathsf{xor}(\mathsf{true}, \mathsf{true}) = \mathsf{false}$, the quotient monad $T\hat{N}$ must identify (x, false) and (x, true) for any $x : X$ (and this suffices to make all computations idempotent).

3. The transformer (T, lift^T) for adding **exceptions** in E is defined as follows, T maps a strong monad $\hat{M}$ to the strong monad $\hat{N}$ given by

   $$\begin{aligned} NX &\triangleq M(X + E) \\ \mathsf{ret}^N_X(x) : NX &\triangleq \mathsf{ret}^M_{X+E}(\mathsf{inl}\, x) \\ \mathsf{throw}_X(e : E) : NX &\triangleq \mathsf{ret}^M_{X+E}(\mathsf{inr}\, e) \\ \mathsf{bind}^N_{X,Y}(c, f) : NY &\triangleq \mathsf{bind}^M_{X+E,Y+E}(c, [f, \mathsf{throw}_X]) \end{aligned}$$

   and lift^T maps $\hat{M}$ to the morphism $\tau : \hat{M} \longrightarrow T\hat{M}$ given by

   $$\tau_X(c : MX) : T\hat{M}X \triangleq \mathsf{bind}^M_{X,X+E}(c, \mathsf{ret}^N_X)$$

   This transformer is functorial (since it is the instance of Example 4.6 with $SX = E$), more precisely T maps an endofunctor F to the endofunctor $F(- + E)$, but it **is not monoidal**. In fact, if it were monoidal, then there should be a natural transformation

   $$\phi_{G,F} : G(F(- + E) + E) \xrightarrow{\bullet} G(F(- + E)).$$

   However, this is impossible when $E = 1$, $GX = X$ and $FX = 0$: in **Set** there is no function $\phi_{G,F} : 1 \longrightarrow 0$.

Lifting Theorem	Algebraic[a] Theorem 3.5	Codensity[b] Theorem 4.15	Monoidal Theorem 4.9
Transformer	Any	Functorial	Monoidal
Operation	$S \longrightarrow\!\!\!\!\!\rightarrow M$	$S \otimes M \longrightarrow\!\!\!\!\!\rightarrow M$	$(S \otimes M) \otimes F \longrightarrow\!\!\!\!\!\rightarrow M$

Figure 4.1: Transformers, operations and lifting theorems.

[a] Algebraic lifting is given for algebraic operations $S \otimes M \longrightarrow\!\!\!\!\!\rightarrow M$ for $\hat{M}$.
[b] Codensity lifting requires a right-closed monoidal category.

4.3 Lifting Through a Transformer

Theorem 3.5 shows how every algebraic operation for a monoid lifts along a monoid morphism. Therefore, given a monoid transformer (T, lift^T) and a monoid $\hat{M}$, every algebraic operation $op : S \otimes M \longrightarrow\!\!\!\!\!\rightarrow M$ for $\hat{M}$ can be lifted along $\mathsf{lift}^T_{\hat{M}}$. We take advantage of the additional structure in functorial and monoidal monoid transformers to provide liftings for more general classes of operations. The results are summarized in Figure 4.1: as one goes from left to right the operations become more general, but the lifting theorems need additional assumptions on the transformers (or the monoidal category, see Theorem 4.15).

To simplify proofs, in the rest of this chapter we assume that $\hat{\mathcal{E}}$ is a strict monoidal category. However, statements and definitions do not rely on this simplifying assumption. We start with a result for monoidal monoid transformers.

Theorem 4.9 (Monoidal Lifting)**.** *If (T, lift^T) is a monoidal monoid transformer with underlying monoidal functor (T, ϕ_I, ϕ), and $\mathsf{op} : S \otimes M \longrightarrow\!\!\!\!\!\rightarrow M$ is a first-order operation for $\hat{M}$, then there is a lifting of op along lift^T_M given by*

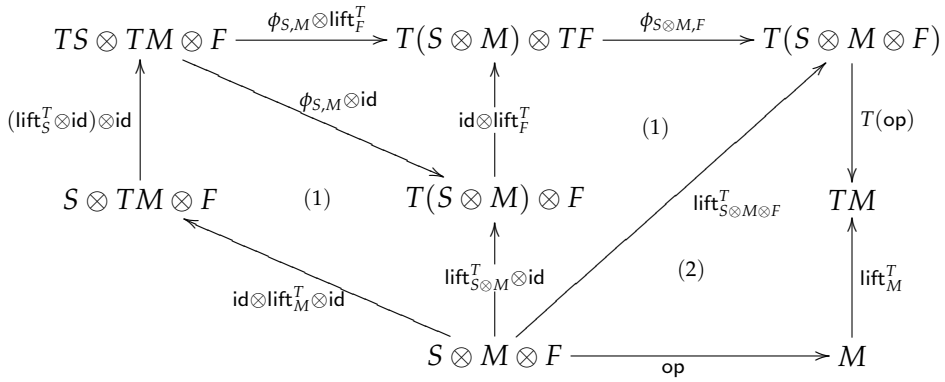
$$\mathsf{op}^T \mathrel{\hat{=}} S \otimes TM \xrightarrow{\mathsf{lift}^T_S \otimes \mathsf{id}} TS \otimes TM \xrightarrow{\phi_{S,M}} T(S \otimes M) \xrightarrow{T(\mathsf{op})} TM$$

More generally, if H is the functor $H(-) = S \otimes U(-) \otimes F$, and $\mathsf{op} : H\hat{M} \longrightarrow\!\!\!\!\!\rightarrow M$ is an H-operation for $\hat{M}$, then there is a lifting op^T of op along lift^T_M given by

$$\begin{array}{ccccc} TS \otimes TM \otimes F & \xrightarrow{\phi_{S,M} \otimes \mathsf{lift}^T_F} & T(S \otimes M) \otimes TF & \xrightarrow{\phi_{S \otimes M, F}} & T(S \otimes M \otimes F) \\ {\scriptstyle (\mathsf{lift}^T_S \otimes \mathsf{id}) \otimes \mathsf{id}} \uparrow & & & & \downarrow {\scriptstyle T(\mathsf{op})} \\ S \otimes TM \otimes F & & \xrightarrow[\mathsf{op}^T]{} & & TM \end{array}$$

Proof. The first-order case reduces to the general case when $F = I$. To show that $\mathsf{op}^T \circ (\mathsf{id} \otimes \mathsf{lift}^T_M \otimes \mathsf{id}) = \mathsf{lift}^T_M \circ \mathsf{op}$ we expand the definition of op^T and prove that the following diagram

commutes



(1) because lift^T is a monoidal natural transformation.

(2) because lift^T is a natural transformation.

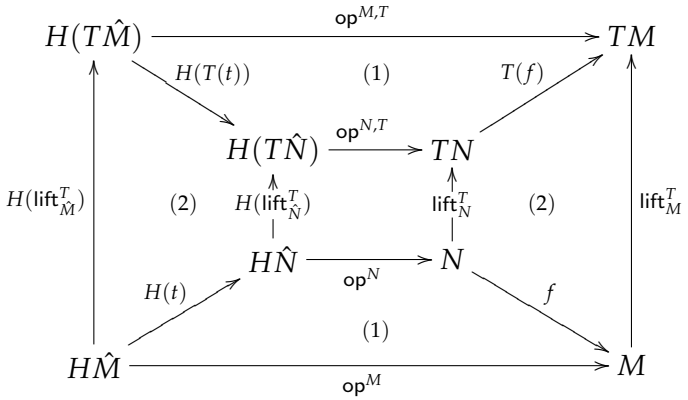
□

We now focus on functorial monoid transformers. Before proving the main result (Theorem 4.15), we establish the following lemma.

Lemma 4.10 (Derived Lifting)**.** *If (T, lift^T) is a functorial monoid transformer, $\mathsf{op}^N : H\hat{N} \longrightarrow N$ is an H-operation for $\hat{N}$, $\mathsf{op}^{N,T} : H(T\hat{N}) \longrightarrow TN$ is a lifting of op^N along $\mathsf{lift}_{\hat{N}}^T$, $t : \hat{M} \longrightarrow \hat{N}$ is a monoid morphism and $f : N \longrightarrow M$ is a map, then*

- $\mathsf{op}^M \mathrel{\hat{=}} H\hat{M} \xrightarrow{H(t)} H\hat{N} \xrightarrow{\mathsf{op}^N} N \xrightarrow{f} M$ *is an H-operation for $\hat{M}$, and*
- $\mathsf{op}^{M,T} \mathrel{\hat{=}} H(T\hat{M}) \xrightarrow{H(T(t))} H(T\hat{N}) \xrightarrow{\mathsf{op}^{N,T}} TN \xrightarrow{T(f)} TM$ *is a lifting of op^M along $\mathsf{lift}_{\hat{M}}^T$.*

Proof. The following diagram commutes



(1) by definition of op^M and $\mathsf{op}^{M,T}$.

(2) because lift^T is a natural transformation.

□

Example 4.11. The H-operation collect of the monad $\hat{M}$ of traces over a monoid $(W, 0, +)$ (see Example 3.10) is:

$$\mathsf{collect}_{A,X}(t : MA, f : MX^{(A\times W)}) \mathrel{\hat{=}} \mathsf{let}\,(y, w) = t\,\mathsf{in}\,(f\,t, w)$$

It can be lifted through any functorial monad transformer $\hat{T} = (T, \mathsf{lift}^T)$ using Lemma 4.10 in the following manner:

- Take $\hat{N}$ to be the side-effect monad of Example 3.9 with states in W.
- Take t to be the following monad morphism:

$$t_X\,(m : MX) : NX \mathrel{\hat{=}} \lambda w : W.\,\mathsf{let}\,(x, w') = m\,\mathsf{in}\,(x, w + w')$$

- Take f to be the following natural transformation:

$$f_X\,(n : NX) : MX \mathrel{\hat{=}} n\,0$$

- $\mathsf{op}^N_{A,X} : (t : NA, f : (NX)^{(A\times W)}) = \mathsf{bind}^N(t, \lambda a : A.\,\mathsf{get}\,(\lambda w : W.\,f\,(a, w)))$
- Since op^N is defined from bind^N and the algebraic operation get, it can be lifted through the monad transformer $\hat{T}$:

$$\mathsf{op}^{N,T}_{A,X} : (t : TNA, f : (TNX)^{(A\times W)}) = \mathsf{bind}^{TN}(t, \lambda a : A.\,\mathsf{get}^T\,(\lambda w : W.\,f\,(a, w)))$$

  where get^T is the algebraic lifting of get.

It can be shown that op^M as defined in the lemma is equivalent to collect, and therefore $\mathsf{op}^{M,T}$ is a lifting of collect through $\hat{T}$.

The example shows that, in order to lift an operation, it may help to express the operation in another monad, even when op^N is not a lifting of op^M along t. It also shows that there are many degrees of freedom that need to be fixed in order to use Lemma 4.10. The Codensity Lifting presented next shows that when the operation is first-order and the underlying category is monoidal right-closed, there is a canonical way to fix these degrees of freedom.

Codensity Lifting

Consider the instance of Lemma 4.10 for $H(-) = S \otimes U(-)$: if op^N is an algebraic operation for $\hat{N}$, then op^M is a first-order operation and one gets a lifting $\mathsf{op}^{M,T}$ of op^M along $\mathsf{lift}^T_{\hat{M}}$ by taking as $\mathsf{op}^{N,T}$ the algebraic lifting of op^N along $\mathsf{lift}^T_{\hat{N}}$. We show that every first-order operation op^M can be defined (as described in Lemma 4.10) using an algebraic operation op^N, provided the monoidal category $\hat{\mathcal{E}}$ is right-closed.

Remark 4.12. In the rest of this section we assume that the strict monoidal category $\hat{\mathcal{E}}$ is right-closed. For example, we could consider $\hat{\mathcal{E}}$ to be the category $\mathsf{Endo}(\mathcal{C})_f$ of finitary functors over a locally finitely presentable category $\mathcal{C}$ of Example 2.9, the category of realisable endofunctors $\mathsf{Endo}(\mathcal{P}_A)_r$ of Example 2.10, or the category of expressible endofunctors $\hat{\mathcal{E}}_F$ of Example 2.11. In this setting, two maps $X \xrightarrow{f_i} G^F$ are equal iff $X \otimes F \xrightarrow{f_i \otimes \mathsf{id}} G^F \otimes F \xrightarrow{ev} G$ are equal.

Definition 4.13 (Codensity)**.** The *codensity* monoid transformer (K, lift^K) is given by

- $K\hat{M} \triangleq (M^M, i_M, c_M)$ is the monoid of endomorphisms of Example 2.16, whose definition is recalled here for the simplified case of a strict monoidal category.

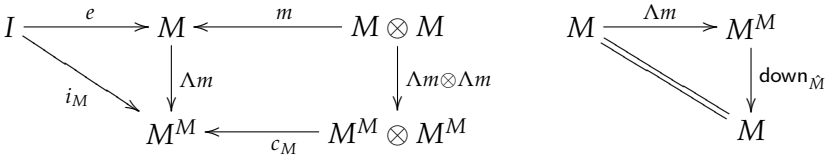
$$i_M \triangleq \Lambda(I \otimes M = M \xrightarrow{\mathsf{id}} M)$$
$$c_M \triangleq \Lambda(M^M \otimes M^M \otimes M \xrightarrow{\mathsf{id} \otimes ev} M^M \otimes M \xrightarrow{ev} M)$$

- $\mathsf{lift}^K_{\hat{M}} \triangleq (M \xrightarrow{\Lambda m} M^M)$ is a monoid morphism $\hat{M} \longrightarrow K\hat{M}$

Moreover, $\mathsf{lift}^K_{\hat{M}}$ has a left inverse $\mathsf{down}_{\hat{M}} : M^M \longrightarrow M$, i.e. $\mathsf{down}_{\hat{M}} \circ \mathsf{lift}^K_{\hat{M}} = \mathsf{id}_M$, given by

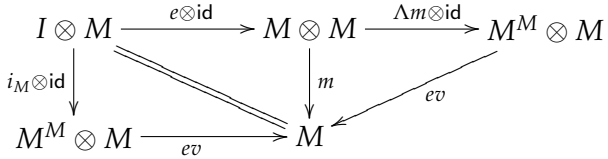
$$\mathsf{down}_{\hat{M}} \triangleq (M^M = M^M \otimes I \xrightarrow{\mathsf{id} \otimes e} M^M \otimes M \xrightarrow{ev} M)$$

Proof. This definition has some proof obligations, i.e.: Λm is a monoid morphism and $\mathsf{down}_{\hat{M}}$ is a left inverse of $\mathsf{lift}^K_{\hat{M}}$. Diagrammatically:

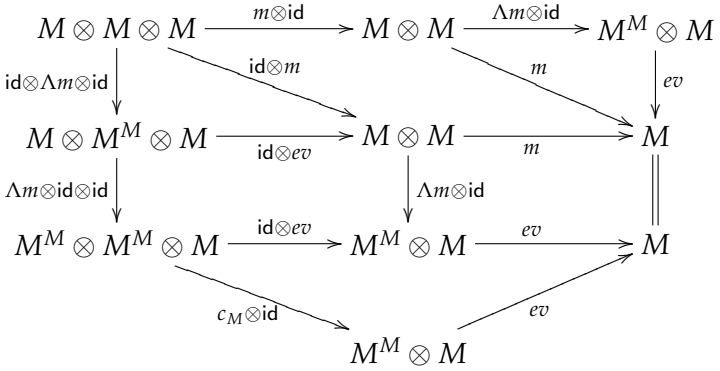


To prove commutativity of the first diagram we use Remark 4.12, the universal property of exponentials, bifunctoriality of $\otimes$ and the monoid laws.

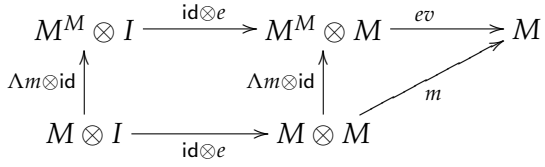
- Λm respects the unit of the monoid.



- Λm respects the multiplication of the monoid.



To prove commutativity of the second diagram we use the definition of $\mathsf{down}_{\hat{M}}$, the universal property of exponentials and the equation $m \circ (\mathsf{id} \otimes e) = \mathsf{id}_M$ for $\hat{M}$.



□

Theorem 4.14 (Codensity Properties)**.** *If* $\mathsf{op} : S \otimes M \longrightarrow M$ *is a first-order operation for* $\hat{M}$*, then*

(a) $\mathsf{op}^K \mathrel{\hat{=}} S \otimes M^M \xrightarrow{\Lambda(\mathsf{op}) \otimes \mathsf{id}} M^M \otimes M^M \xrightarrow{c_M} M^M$ *is algebraic for* $K\hat{M}$

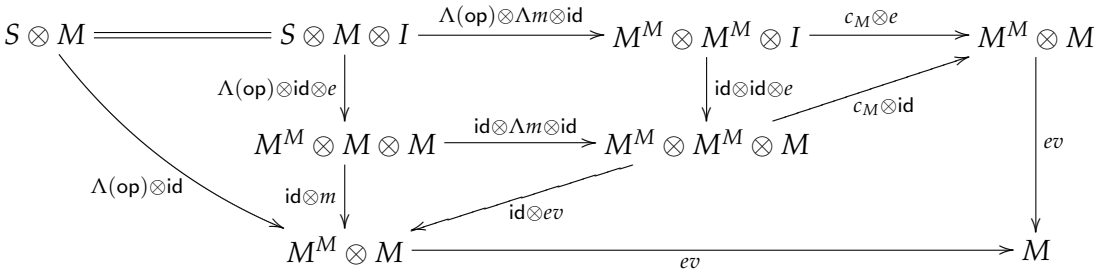
(b) $\mathsf{op} = S \otimes M \xrightarrow{\mathsf{id} \otimes \mathsf{lift}^K_{\hat{M}}} S \otimes M^M \xrightarrow{\mathsf{op}^K} M^M \xrightarrow{\mathsf{down}_{\hat{M}}} M$

Moreover, if op *is algebraic, then* op^K *is the algebraic lifting of* op *along* $\mathsf{lift}^K_{\hat{M}}$.

Proof. The operation op^K is the algebraic operation induced by $S \xrightarrow{\Lambda(\mathsf{op})} M^M$ (see Proposition 3.3), hence item (a) is proved. In order to prove item (b), we expand the definitions and the equation becomes:

$$\mathsf{op} = S \otimes M \xrightarrow{\Lambda(\mathsf{op}) \otimes \Lambda m} M^M \otimes M^M \xrightarrow{c_M \otimes e} M^M \otimes M \xrightarrow{ev} M$$

and the proof is given by the following commuting diagram (see Remark 4.12).



Finally, if op is algebraic, then $\mathsf{op} = S \otimes M \xrightarrow{\mathsf{op}' \otimes \mathsf{id}} M \otimes M \xrightarrow{m} M$ for a unique map $\mathsf{op}' : S \longrightarrow M$ (see Proposition 3.3). Therefore

$$\Lambda(\mathsf{op}) = S \xrightarrow{\mathsf{op}'} M \xrightarrow{\Lambda m} M^M$$

and op^K is the algebraic operation induced by the lifting of op' along $\mathsf{lift}^K_{\hat{M}}$. □

We now state our main lifting result for functorial monoid transformers.

Theorem 4.15 (Codensity Lifting)**.** *Given a functorial monoid transformer (T, lift^T) and a first-order operation $\mathsf{op} : S \otimes M \longrightarrow M$ for a monoid $\hat{M}$, there is a lifting of op along $\mathsf{lift}^T_{\hat{M}}$ given by*

$$\mathsf{op}^T \mathrel{\hat{=}} S \otimes TM \xrightarrow{\mathsf{id} \otimes T(\mathsf{lift}^K_{\hat{M}})} S \otimes T(M^M) \xrightarrow{\mathsf{op}^{K,T}} T(M^M) \xrightarrow{T(\mathsf{down}_{\hat{M}})} TM$$

where $\mathsf{op}^{K,T}$ is the unique algebraic lifting of op^K along $\mathsf{lift}^T_{K\hat{M}}$.

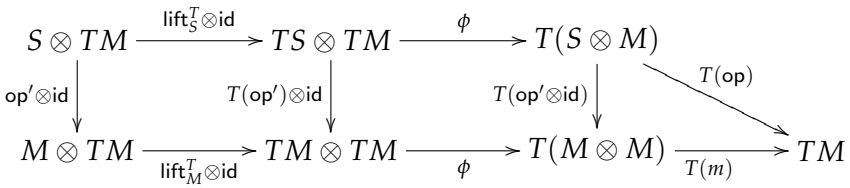
Proof. Apply Lemma 4.10 by taking $\mathsf{op}^M = \mathsf{op}$, $\hat{N} = K\hat{M}$, $\mathsf{op}^N = S \otimes N \xrightarrow{\mathsf{op}^K} N$, thus op^N is algebraic for $\hat{N}$ (by Theorem 4.14), $t = \mathsf{lift}^K_{\hat{M}}$, $f = \mathsf{down}_{\hat{M}}$, and $\mathsf{op}^{N,T} : S \otimes (TN) \longrightarrow TN$ the unique algebraic lifting of op^N along $\mathsf{lift}^T_{\hat{N}}$. □

4.4 Coincidence of Liftings

For some combinations of monoid transformers and operations it is possible that two (or more) of the lifting theorems summarized in Figure 4.1 are applicable. For instance, if op is an algebraic operation for $\hat{M}$ and (T, lift^T) is a monoidal monoid transformer, then one can apply both the algebraic lifting (Theorem 3.5) and the monoidal lifting (Theorem 4.9). In this section it is proved that when more than one of the lifting theorems is applicable, they yield the same result.

Theorem 4.16 (Algebraic/Monoidal Coincidence)**.** *When (T, lift^T) is a monoidal monoid transformer, and $\mathsf{op} : S \otimes M \longrightarrow M$ is an algebraic operation for $\hat{M}$, the monoidal lifting (Theorem 4.9) and the algebraic lifting (Theorem 3.5) of op along $\mathsf{lift}^T_{\hat{M}}$ coincide.*

Proof. Since op is an algebraic operation for $\hat{M} = (M, e, m)$, by Proposition 3.3 there exists a unique $\mathsf{op}' : S \longrightarrow M$ such that $\mathsf{op} = m \circ (\mathsf{op}' \otimes \mathsf{id})$. Consider the following diagram, where the top path from $S \otimes TM$ to TM is the monoidal lifting of op, and the bottom path is the algebraic lifting of op. The diagram commutes because of naturality of lift^T and ϕ.



□

Theorem 4.17 (Algebraic/Codensity Coincidence)**.** *When* (T, lift^T) *is a functorial monoid transformer (on a monoidal right-closed category), and* $\mathsf{op} : S \otimes M \longrightarrow M$ *is an algebraic operation for* $\hat{M}$*, the codensity lifting (Theorem 4.15) and the algebraic lifting (Theorem 3.5) of* op *along* $\mathsf{lift}^T_{\hat{M}}$ *coincide.*

Proof. Since op is an algebraic operation for $\hat{M} = (M, e, m)$, by Proposition 3.3 there exists a unique $\mathsf{op}' : S \longrightarrow M$ such that $\mathsf{op} = m \circ (\mathsf{op}' \otimes \mathsf{id})$. Similar characterizations hold for the following algebraic operations:

- $\mathsf{op}^T : S \otimes TM \longrightarrow TM$ is the algebraic lifting of op along $\mathsf{lift}^T_{\hat{M}}$, therefore it is algebraic for the monoid $T\hat{M}$ and corresponds to $\mathsf{lift}^T_{\hat{M}} \circ \mathsf{op}'$;
- $\mathsf{op}^K : S \otimes M^M \longrightarrow M^M$ (see Proposition 4.14) is the algebraic lifting of op along $\mathsf{lift}^K_{\hat{M}}$, therefore it is algebraic for the monoid $K\hat{M}$ and corresponds to $\mathsf{lift}^K_{\hat{M}} \circ \mathsf{op}' = \Lambda(\mathsf{op})$;
- $\mathsf{op}^{K,T} : S \otimes T(M^M) \longrightarrow T(M^M)$ given by the algebraic lifting of op^K along $\mathsf{lift}^T_{K\hat{M}}$ is algebraic for the monoid $T(K\hat{M})$ and corresponds to $\mathsf{lift}^T_{K\hat{M}} \circ \Lambda(\mathsf{op})$.

By naturality of lift^T, $T(\mathsf{lift}^K_{\hat{M}}) \circ \mathsf{lift}^T_{\hat{M}} = \mathsf{lift}^T_{K\hat{M}} \circ \mathsf{lift}^K_{\hat{M}}$, thus $\mathsf{op}^{K,T}$ is the algebraic lifting of op^T along $T(\mathsf{lift}^K_{\hat{M}})$ and the following diagram commutes (the bottom path from $S \otimes TM$ to TM is the codensity lifting of op). The triangle commutes because of functoriality of T and because $\mathsf{down}_{\hat{M}}$ is a left inverse of $\mathsf{lift}^K_{\hat{M}}$ (see Definition 4.13).

$$\begin{array}{ccccc} S \otimes TM & & \xrightarrow{\mathsf{op}^T} & & TM \\ {\scriptstyle \mathsf{id} \otimes T(\mathsf{lift}^K_{\hat{M}})}\downarrow & & & {\scriptstyle T(\mathsf{lift}^K_{\hat{M}})}\downarrow & \\ S \otimes T(M^M) & \xrightarrow[\mathsf{op}^{K,T}]{} & T(M^M) & \xrightarrow[T(\mathsf{down}_{\hat{M}})]{} & TM \end{array}$$

□

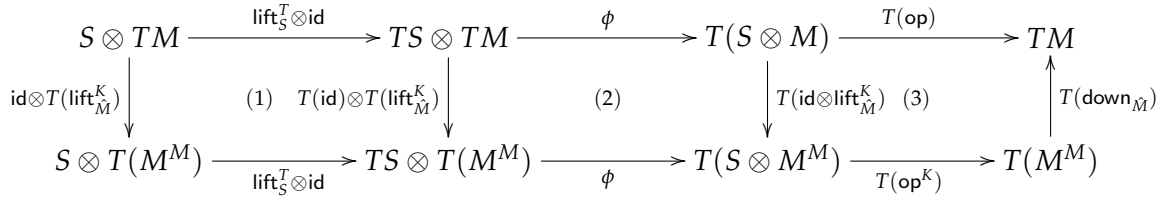
Theorem 4.18 (Codensity/Monoidal Coincidence)**.** *When* (T, lift^T) *is a monoidal monoid transformer (on a monoidal right-closed category), and* $\mathsf{op} : S \otimes M \longrightarrow M$ *is a first-order operation for* $\hat{M}$*, the codensity lifting (Theorem 4.15) and the monoidal lifting (Theorem 4.9) of* op *along* $\mathsf{lift}^T_{\hat{M}}$ *coincide.*

Proof. The codensity lifting of op is given by

$$S \otimes TM \xrightarrow{\mathsf{id} \otimes T(\mathsf{lift}^K_{\hat{M}})} S \otimes T(M^M) \xrightarrow{\mathsf{op}^{K,T}} T(M^M) \xrightarrow{T(\mathsf{down}_{\hat{M}})} TM$$

where $\mathsf{op}^{K,T}$ is the algebraic lifting of the algebraic operation op^K along $\mathsf{lift}^T_{K\hat{M}}$, or equivalently (by Theorem 4.16), $\mathsf{op}^{K,T}$ is the monoidal lifting of op^K along $\mathsf{lift}^T_{K\hat{M}}$.

Consider the following diagram, where the top path from $S \otimes TM$ to TM is the monoidal lifting of op, and the bottom path is the codensity lifting of op



The diagram commutes for the following reasons:

(1) by bifunctoriality of $\otimes$;

(2) because ϕ is a natural transformation;

(3) by item (b) of Theorem 4.14 (and functoriality of T).

□

4.5 Examples of Lifted Operations

In this section, the Codensity Lifting (Theorem 4.15) is specialized to various concrete monad transformers and an arbitrary first-order operation $\mathsf{op} : \Sigma \otimes M \longrightarrow M$ over a monoid $\hat{M}$. The obtained liftings of op show that the Codensity Lifting subsumes the incremental approach in (Benton, Hughes, and Moggi 2000; Moggi 1997). Additionally, each lifting is instantiated to the non-algebraic first-order operations local, handle, and flush, showing that the obtained lifted operations are the "expected" ones, in the sense that they coincide with the existing ad-hoc liftings in the literature.

- When the functorial monad transformer $\hat{T}$ is the side-effect monad transformer, thus $TMX = M(X \times S)^S$, the lifting simplifies to:

$$\mathsf{op}_X^T\,(t : \Sigma(TMX)) : TMX = \lambda s : S.\, \mathsf{op}_{X \times S}(\mathsf{map}^\Sigma\, \tau^s\, t)$$

  where $\tau^s(f : (M(X \times S))^S) \triangleq f\, s$.

  - When $\hat{M}$ is the monad for environments in V, thus $TMX = ((X \times S)^V)^S$, the lifting of local yields:

$$\mathsf{local}^T\,(f : V^V, t : TMX) : TMX = \lambda s : S\,.\lambda v : V.\, t\, s\, (f\, v).$$

  - When $\hat{M}$ is the monad for exceptions in E, thus $TMX = ((X \times S) + E)^S$, the lifting of handle yields:

$$\mathsf{handle}^T\,(t : TMX, h : (TMX)^E) : TMX = \lambda s : S.\, \mathsf{case}\ t\, s\ \mathsf{of}\ \mid \mathsf{inl}\, e \Rightarrow h\, e\, s \mid \mathsf{inr}\, x \Rightarrow \mathsf{inr}\, x.$$

– When $\hat{M}$ is the monad for traces over a monoid $(W, 0, +)$, thus $TMX = ((X \times S) \times W)^S$, the lifting of flush yields:

$$\mathsf{flush}^T\,(t : TMX) : TMX = \lambda s : S.\,\mathsf{let}\ (x : X, w : W) = t\,s\ \mathsf{in}\ (x, 0).$$

- When the functorial monad transformer $\hat{T}$ is the exception monad transformer, thus $TMX = M(X + E)$, the lifting simplifies to:

$$\mathsf{op}^T_X(t : \Sigma(TMX)) : TMX = \mathsf{op}_{X+E}\,t.$$

– When $\hat{M}$ is the monad for environments in S, thus $TMX = (X + E)^S$, the lifting of local yields:

$$\mathsf{local}^T\,(f : S^S, t : TMX) : TMX = \lambda s : S.\,t\,(f\,s).$$

– When $\hat{M}$ is the monad for exceptions in E', thus $TMX = (X + E) + E'$, the lifting of handle yields:

$$\mathsf{handle}^T(t : TMX, h : (TMX)^{E'}) : TMX = \mathsf{case}\ t\ \mathsf{of}\ \mid \mathsf{inl}\,e \Rightarrow h\,e \mid \mathsf{inr}\,x \Rightarrow \mathsf{inr}\,x.$$

– When $\hat{M}$ is the monad for traces over a monoid $(W, 0, +)$, thus $TMX = (X + E) \times W$, the lifting of flush yields:

$$\mathsf{flush}^T\,((c, w) : TMX, h : (TMX)^E) : TMX = (c, 0).$$

- When the functorial monad transformer $\hat{T}$ is the functorial monad transformer for environments in S, thus $TMX = (MX)^S$, the lifting simplifies to:

$$\mathsf{op}^T_X\,(t : \Sigma(TMX)) : TMX = \lambda s : S.\,\mathsf{op}_X(\mathsf{map}^\Sigma\,\tau^s\,t)$$

where $\tau^s(f : (MX)^S) = f\,s$.

– When $\hat{M}$ is the monad for environments in S', thus $TMX = (X^{S'})^S$, the lifting of local yields:

$$\mathsf{local}^T\,(f : S'^{S'}, t : TMX) : TMX = \lambda s : S.\,\lambda s' : S'.\,t\,s\,(f\,s').$$

– When $\hat{M}$ is the monad for exceptions in E, thus $TMX = (X + E)^S$, the lifting of handle yields:

$$\mathsf{handle}^T\,(t : TMX, h : (TMX)^E) : TMX = \lambda s : S.\,\mathsf{case}\ t\,s\ \mathsf{of}\ \mid \mathsf{inl}\,e \Rightarrow h\,e\,s \mid \mathsf{inr}\,x \Rightarrow \mathsf{inr}\,x.$$

– When $\hat{M}$ is the monad for traces over a monoid $(W, 0, +)$, thus $TMX = (X \times W)^S$, the lifting of flush yields:

$$\mathsf{flush}^T\,(t : TMX) : TMX = \lambda s : S.\,\mathsf{let}\,(x : X, w : W) = t\,s\,\mathsf{in}\,(x, 0).$$

- When the functorial monad transformer $\hat{T}$ is the functorial monad transformer for traces over a monoid $(W, 0, +)$, thus $TMX = M(X \times W)$, the lifting simplifies to:

$$\mathsf{op}^T_X(t : \Sigma(TMX)) : TMX = \mathsf{op}_{X\times W}\,t.$$

– When $\hat{M}$ is the monad for environments in S, thus $TMX = (X \times W)^S$, the lifting of local yields:

$$\mathsf{local}^T\,(f : S^S, t : TMX) : TMX = \lambda s : S.\,t\,(f\,s).$$

– When $\hat{M}$ is the monad for exceptions in E, thus $TMX = (X \times W) + E$, the lifting of handle yields:

$$\mathsf{handle}^T(t : TMX, h : (TMX)^E) : TMX = \mathsf{case}\ t\ \mathsf{of}\ \mid \mathsf{inl}\,e \Rightarrow h\,e \\ \mid \mathsf{inr}\,x \Rightarrow \mathsf{inr}\,x.$$

– When $\hat{M}$ is the monad for traces over a monoid $(W', 0', +')$, thus $TMX = (X \times W) \times W'$, the lifting of flush yields:

$$\mathsf{flush}^T\,((p, w') : TMX) : TMX = (p, 0').$$

4.6 Summary

We have defined a hierarchy of monoid transformers and shown several general liftings results that are applicable to wide classes of operations. Moreover, the obtained liftings have been shown to coincide when more than one of them is applicable to the same operation. Using these results, all the operations considered by Liang, Hudak, and Jones (1995) and all the operations in the mtl can be lifted through any functorial monad transformer. Through several examples, we have given evidence that our uniform lifting subsumes the more or less ad-hoc definitions of lifting that could be found in the literature.

The theoretical foundation for lifting of operations presented in this chapter was formulated using category theory. This makes the results very general and susceptible to be applied to other structures apart from monads by different instantiations of the monoidal category. In this thesis, however, the focus is in the instantiation of the theory in functor categories, where monoids are monads, and monoid transformers are monad transformers.

Monad transformers were first considered by Moggi (1989*a*) and called *monad constructors*. The associated operations were considered to be morphisms between first-order types.

A notion of natural lifting was defined and a few results about lifting were given. Some of these ideas were implemented in Scheme by Espinosa (1993) (see also Espinosa 1995). Liang, Hudak, and Jones (1995) implemented monad transformers in a strongly-typed language for the first time and extended the class of operations considered by Moggi by incorporating higher-order types. Also, ad-hoc liftings not considered by Moggi were provided for several computational effects. Moggi (1997) (see also Benton, Hughes, and Moggi 2000) provided liftings for a generic notion of operation, but for a fixed transformer, obtaining results similar to the ones in Section 4.5, except that the results presented here all stem from a single uniform lifting instantiated to different transformers.

Some of the related work focuses on the combination of monads but the problem of lifting operations is not tackled. King and Wadler (1992) discussed some issues in composing monads. Jones and Duponcheel (1993) analyzed functorial composition of monads, which inevitably led them to distributive laws of monads (Barr and Wells 1985). They showed that the list monad can be combined with any other *commutative* monad by functorial composition. In Example 4.7 it was shown how to define a list transformer that lifts the commutativity restriction, but the combination is more subtle than functorial composition. Lüth and Ghani (2002) implemented the coproduct of monads. This construction is very general and works on a wide class of monads, but the ordering of effects is not taken into account (coproducts are commutative up to isomorphism).

In the algebraic view of computational effects advocated by Hyland, Plotkin, and Power (2006) (see also Plotkin and Power 2001*a*,*b*, 2002, 2004), the focus is put on operations and equations. Instead of modelling computational effects with monads, one considers algebraic theories, and computational effects are composed by combining algebraic theories. Operations which are not algebraic are not supported, but some of these operations are handlers of algebraic effects, which can be understood as homomorphisms from the free model of the theory (Plotkin and Pretnar 2009). However, it remains to be shown how to lift a handler to a combined theory. A limitation of the algebraic view is that notions like continuations, which do not arise from algebraic theories, are not tractable within this approach.

Part II

Applications

Chapter 5

Monatron: A Monad Transformer Library

In this chapter we apply the theory developed in the first part of this thesis to the problem of implementing a monad transformer library in Haskell. The implementation of the theory in Haskell is reasonably straightforward, but in order to have a good library there are other requirements that need to be met. Hence, in this chapter, we will mostly focus on the design decisions that were made during the implementation.

In the first section we review how monads, functors and monad transformers are implemented in Haskell. Then, in Section 5.2, we analyze some shortcomings of current monad transformer libraries, such as mtl (*Monad Transformer Library*). In particular, it is shown that the lifting of operations through monad transformers is done on a case-by-case basis, and consequently there is no guarantee that the liftings are uniform, that extending the library is cumbersome, that some liftings cannot be expressed because the lifting overloading mechanism produces shadowing of operations, and that the design relies essentially on non-portable features of type classes.

The mtl has been distributed with GHC (*Glasgow Haskell Compiler*) and is now part of the the *Haskell Platform* (2009), a collection of unessential but widely used libraries. It can be considered the *de facto* standard of monad transformer libraries for Haskell.

A first attempt to remedy the problems in the mtl was to incorporate uniform liftings to the mtl. This lead to the implementation of mmtl (*Modular Monad Transformer Library*), which is almost[1] a drop-in replacement for the mtl, but adds uniform liftings. The library mmtl was implemented with the goal of obtaining backward-compatibility with the mtl while adding uniform liftings but, in doing this, it carried over to the new library design flaws in the mtl. Monatron is our attempt to remedy all these flaws, and its design is discussed in Section 5.3.

The complete source code of the Monatron library is included in Appendix A.

[1] Some liftings in the mtl, such as callCC through StateT are different.

```
data Either x a = Left x | Right a
instance Functor (Either x) where
  fmap f (Left x)  = Left x
  fmap f (Right a) = Right (f a)
instance Monad (Either x) where
  return a       = Right a
  Left x  >>= f = Left x
  Right a >>= f = f a
```

```
newtype Id a = Id a
instance Functor Id where
  fmap f (Id a) = Id (f a)
instance Monad Id where
  return a      = Id a
  (Id a) >>= f = f a
```

Figure 5.1: Either and Id and their Monad and Functor instances

5.1 Functors, Monads and Monad Transformers in Haskell

In Haskell, a datatype constructor of kind $* \to *$ is shown to have a functorial or a monadic structure by instances of the following type classes.

```
class Functor f where
  fmap :: (a → b) → f a → f b
class Monad m where
  return :: a → m a
  (>>=) :: m a → (a → m b) → m b
```

Instances of Functor are required to preserve identities and composition:

$$\begin{aligned} \mathsf{fmap}\ \mathsf{id} &= \mathsf{id} \\ \mathsf{fmap}\ (f \cdot g) &= \mathsf{fmap}\ f \cdot \mathsf{fmap}\ g \end{aligned}$$

Instances of the Monad class are required to satisfy the following equations that ensure that return and $(\gg\!=)$ are well-behaved:

$$\begin{aligned} \mathsf{return}\ a \gg\!= f &= f\ a \\ m \gg\!= \mathsf{return} &= m \\ m \gg\!= (\lambda a \to f\ a \gg\!= g) &= (m \gg\!= f) \gg\!= g \end{aligned}$$

Satisfaction of these equations cannot be verified by the type-checker so it is the responsibility of the programmer to verify the correctness of each instance.

As an example, in Figure 5.1, we have defined two datatype constructors and given them their corresponding Functor and Monad instances: Either x is a monad for exceptions of type x, and the identity monad Id is a monad of pure computations.

Combinator libraries for monads come equipped with several standard monads corresponding to different computational effects that provide readily available building blocks for constructing effectful computations. For example, libraries usually provide a State s

monad for modelling side-effects of type s, a Cont r monad for modelling continuations with result type r, a Writer w monad for modelling traces over a monoid w, a Reader e monad for modelling environments of type e, and an Exception x monad for modelling exceptions of type x. These monads provide some of the most common computational effects but, by all means, they are not the only ones. The fact that monad libraries (mostly) only support this limited set of effects and have not been extended to other effects can be seen as evidence of the universality of these effects. Less optimistically, it can be seen as symptomatic of a lack of extensibility (see Section 5.2).

In addition to the aforementioned monads, combinator libraries provide the corresponding monad transformer for each effect. A monad transformer in Haskell is a type constructor of kind $(* \to *) \to (* \to *)$ which is an instance of the type class MonadT.

```
class MonadT t where
  treturn :: Monad m ⇒ a → t m a
  treturn = lift · return
  tbind   :: Monad m ⇒ t m a → (a → t m b) → t m b
  lift    :: Monad m ⇒ m a → t m a
```

Instances of MonadT are required to make treturn and tbind satisfy the monad laws (making $t\ m$ a monad for every monad m) and to make lift a *monad morphism* from m into $t\ m$, i.e. lift should preserve return and $(\gg\!=)$:

$$\mathsf{treturn} = \mathsf{lift} \cdot \mathsf{return} \tag{5.1}$$

$$\mathsf{lift}\ (m \gg\!= f) = \mathsf{lift}\ m\ \text{`tbind`}\ (\mathsf{lift} \cdot f) \tag{5.2}$$

The type class MonadT provides a default implementation of treturn given by the equation (5.1), and therefore its instances only need to provide definitions for tbind and lift.

In most Haskell libraries, lift is the only member of the type class, and every instance T of MonadT is required to provide an instance:

```
instance Monad m ⇒ Monad (T m) where
  ...
```

However this assumption cannot be expressed in the type class MonadT and the equations that lift must satisfy need to have a side condition. With our formulation, the required equations are expressible in terms of members of the type class, maintaining consistency with the definition of other categorical constructs in Haskell such as monads and functors. The disadvantage of our approach is that the syntactic sugar for monadic computations available in Haskell is lost: the **do** notation works for return and $\gg\!=$ and not for treturn and tbind. This is easily solved by adding the appropriate monad instance for each monad transformer T:

```
instance Monad m ⇒ Monad (T m) where
  return = treturn
  (>>=) = tbind
```

Although it would be more efficient to provide one general instance for an arbitrary monad transformer, this would result in overlapping instances.

Since a monad transformer adds an effect to another monad, monads such as the side-effect monad State are equivalent to adding side-effects to the identity monad Id.

$$\mathsf{State}\ s \equiv \mathsf{StateT}\ s\ \mathsf{Id}$$

Therefore, in principle, only the transformer version of a monad is needed, and concrete monads will be defined as an instance of the monad transformer at the identity monad, avoiding repetition of instances for the same computational effect and ensuring consistency. For example:

```
type State s = StateT s Id
```

Importantly, for the combined monads, the programmer has no need to verify any monad laws, or declare new monad instances. The monad transformers will guarantee that the obtained type is a monad by construction, realising the idea that correct constructions are obtained by combining correct components.

Encapsulation of Effects

When declaring a new datatype in Haskell one has the option of defining it as a type synonym (using the keyword **type**), as a datatype with constructor functions (using the keyword **data**), or using the keyword **newtype** which is a datatype with only one constructor.

In order to be able to use Haskell's type-class system, monad transformers are defined as **data** or **newtype**. In our case we will also insist on making these types *opaque*. That is, modules implementing the datatype will only export the type but not the constructors, effectively making the monad operations and the associated effect-manipulating operations the only way to construct monadic computations.

This is a common software engineering practice: hide the implementation so that the interface of the library is not changed when the implementation is updated. Possible modifications that should remain hidden to the users of the library could be to replace the list monad with more efficient implementations such as Hughes lists (Hughes 1986) or to replace a monad by a continuation passing style representation of the monad (Filinski 1994).

Despite the obvious benefits of opaque datatypes, most existing libraries expose the internal structure of each monad/monad transformer.

Running Effects

Monad transformers and monads expressible in Haskell come equipped with a *run function* that allows programmers to evaluate an effectful computation. For example, the state monad can be run with $\mathsf{runState} :: s \to \mathsf{State}\ s\ a \to (a, s)$, which given an initial state and a stateful computation, returns the value of the computation together with the final state. The state monad transformer can be run with $\mathsf{runStateT} :: \mathsf{Monad}\ m \Rightarrow s \to \mathsf{StateT}\ s\ m\ a \to m\ (a, s)$, which given an initial state and a computation, returns an m-computation with a value and final state. The exception monad is run with a function $\mathsf{runException} :: \mathsf{Exception}\ x\ a \to \mathsf{Either}\ x\ a$, and the exception monad transformer is run with a function $\mathsf{runExcT} :: \mathsf{Monad}\ m \Rightarrow \mathsf{ExcT}\ x\ m\ a \to m\ (\mathsf{Either}\ x\ a)$, which returns an m-computation over either an exception or a value.

Consider now the following monads that combine side effects and exceptions.

$$\textbf{type}\ \mathsf{StExc}\ s\ x = \mathsf{StateT}\ s\ (\mathsf{Exception}\ x)$$
$$\textbf{type}\ \mathsf{ExcSt}\ x\ s = \mathsf{ExcT}\ x\ (\mathsf{State}\ s)$$

The monad $\mathsf{StExc}\ s\ x$ is obtained by applying the side-effect monad transformer to the exception monad and $\mathsf{ExcSt}\ x\ s$ is obtained by applying the exception monad transformer to the side-effect monad. Are these two monads equivalent? After all, both monads model side-effects together with exceptions. The answer is no, and in general, the order in which the monad transformers are applied is important. To see why, it is necessary to run the effectful computations. The run function for a combined monad is obtained by composing the run functions of its components:

$$\mathsf{runStExc}\ \ :: s \to \mathsf{StExc}\ s\ x\ a \to \mathsf{Either}\ x\ (a, s)$$
$$\mathsf{runStExc}\ s = \mathsf{runException} \cdot \mathsf{runStateT}\ s$$
$$\mathsf{runExcSt}\ \ :: s \to \mathsf{ExcSt}\ x\ s\ a \to (\mathsf{Either}\ x\ a, s)$$
$$\mathsf{runExcSt}\ s = \mathsf{runState}\ s \cdot \mathsf{runExcT}$$

Analysing the type of the resulting run functions, we can see that in StExc, when an exception is raised the computation forgets about the state, while in ExcSt, when an exception is raised the computation preserves the state. One can then choose how exceptions should interact with side-effects by choosing the order in which the monad transformers are applied. In general, applying monad transformers in different orders gives rise to different interactions between effects (Liang, Hudak, and Jones 1995).

5.2 Some Problems with the Traditional Design

The current design of monad transformer libraries performs the liftings of operations of an underlying monad to the transformed monad in an ad-hoc fashion, relying crucially on a type-class trick in order to perform the liftings. The basic idea of the trick is to define a

class of monads supporting a certain operation. For example, we define the class of monads supporting the operation callCC, and show that the continuation monad transformer ContT applied to any monad m is an instance of this class:

```
class Monad m ⇒ ContM m where
  callCC :: ((a → m b) → m a) → m a
instance Monad m ⇒ ContM (ContT m) where
  callCC = ...
```

The final step is to show that, for each monad transformer in the library, if the underlying monad supports callCC, then the transformed monad also supports callCC. For example, for the exception monad transformer ExcT:

```
instance ContM m ⇒ ContM (ExcT x m) where
  callCC = ...
```

This type-class trick has some advantages such as an overloading of the operations that is usually convenient, but libraries that rely on it have some shortcomings which affect the predictability, extensibility, expressive power and portability of the library. In what follows, we explain why this is so.

Non-uniform liftings

One can replace a computation on the Writer w monad over a monoid w by a computation on the more general State w monad, and replace the trace operation trace of the Writer w monad by a trace operation on State w. Similarly, one can do the same replacements on the transformer version of these monads. For example, when the monoid is the monoid of Strings (with unit the empty string `""`, and multiplication given by string concatenation $+\!\!+$), one can replace the operation trace :: Monad m ⇒ String → WriterT String m () that adds a string to a trace, with the following operation:

```
traceS  :: Monad m ⇒ String → StateT String m ()
traceS w = do s ← get
              put (s ++ w)
```

One would expect that replacing WriterT by StateT, replacing trace by trace^S, and replacing runWriterT by runStateT `""`, the semantics of a program would be preserved. However, in the mtl (*Monad Transformer Library*), the following two programs, which perform computations over WriterT String (Cont (String, String)) and StateT String (Cont (String, String)) respectively, have different behaviours:

```
p1 :: (String, String)
p1 = (runCont id · runWriterT)
       (callCC (λexit → trace "1" >>= λ_ → exit "Exit"))
```

```
p2 :: (String, String)
p2 = (runCont id · runStateT "")
       (callCC (λexit → trace^S "1" >>= λ_ → exit "Exit"))
```

While `p1 = ("Exit","")`, we have that `p2 = ("Exit","1")`. The difference in behaviour is caused by non-uniform liftings in the mtl. In particular, callCC is lifted through the StateT monad transformer in a way which is not coherent with the lifting of callCC through WriterT. Although we can regard this as a bug and change the implementation of the library, each operation is lifted on a case-by-case basis and, consequently, there is no intrinsic guarantee that the liftings are coherent. Hence, with no guarantee that the liftings are coherent, the predictability of the semantics of the library is seriously affected.

Quadratic number of instances

Suppose a programmer wants to extend the library with a new monad transformer which comes equipped with some operations. The programmer must write a new class corresponding to the added operations and an instance of this new class for *each* existing monad transformer, so that the added operations can be lifted through other monad transformers. Furthermore, the programmer is required to write instances of *each* existing class of operations for the new monad transformer. In other words, assuming one class of operations per monad transformer, the number of instances increases quadratically with the number of monad transformers.

The extensibility of the library is affected not only because of the quadratic growth of required lines of code, but also because of the lack of separation of concerns. Extending the library requires understanding the semantics of all the existing monad transformers and their operations.

The quadratic growth in the number of instances and lack of separation of concerns is a major hurdle. It discourages anyone willing to extend the library and it shows that the traditional design can only work for a library with a very limited number of monads.

Shadowing of operations

We have seen in Section 5.1 how StExc and ExcST give rise to different interactions between exceptions and state. With the former, state changes are lost when an exception is raised, while with the latter state changes are preserved. Suppose that we need both types of exception. We can easily construct such a monad as follows:

```
type ExcStExc x1 s x2 = ExcT x1 (StExc s x2)
runExcStExc :: s → ExcStExc x1 s x2 a → Either x2 (Either x1 a, s)
runExcStExc s = runStExc s · runExcT
```

We now have two different types of exceptions with two different types of exception (*x1* and *x2*). Let us assume that both *x1* and *x2* are of the same type, say Int. Since there is no type that will distinguish instances, the function handle will refer to the instance of the outermost monad. We have no way of handling the other type of exceptions, as there is no way of saying "What I mean is the handle operation corresponding to the monad under two monad transformers". The inner handle operation is shadowed by the outer one.

One way to deal with this problem is to define different types of exceptions, but this entails inserting unnecessary constructors and destructors which clutter the program and make it more difficult to understand. We see shadowing as revealing a limitation in the expressive power of the library.

Portability

The implementation of the type-class trick requires several extensions to the Haskell 98 standard (Peyton Jones 2003), such as functional dependencies and multi-parameter classes. Although many of these extensions have been around for a while and their semantics are quite stable, they certainly affect the portability of the library. This is particularly so because the whole implementation of the library revolves around this type-class trick. We would like to have the *choice* of paying the price and using the overloading of operations provided by the implementation with type classes when it is convenient without being *forced* to do so.

5.3 The Monatron Approach

We now present the design of the Monatron monad transformer library, which is based on the following ideas:

- Uniform liftings of operations: we implement the uniform liftings explained in Chapters 3 and 4 and summarised in Figure 4.1. For this, we define different type classes that correspond to each type of monad transformer, and define a lifting function for the corresponding class of operations.

- Opaque datatypes. The library provides an interface in which the details of the implementation of monad transformers are hidden. Changes in the implementation of the library should be transparent to its users.

- Explicit liftings. We provide explicit lifting functions that perform the liftings through appropriate monad transformers. This solves the shadowing problem, and removes the forced dependency on many extensions to Haskell 98.

Uniform Liftings

In order to incorporate uniform liftings we need to develop the infrastructure of the first part of the thesis. Many of the results rely on natural transformations, which in Haskell are expressed by polymorphic functions $\forall\ a.\ \mathsf{F}\ a \to \mathsf{G}\ a$, between Functors F and G. This means that the extension to Haskell 98 adding rank-2 types will be essential. However, this is the only extension that will be needed to get a fully functional monad transformer library.

We define the type class of functorial monad transformer FMonadT and the type class of monoidal monad transformer MMonadT. Instances of FMonadT are required to be instances of MonadT and instances of MMonadT are required to be instances of FMonadT, reflecting the inclusions in the monoid transformer hierarchy of Section 4.1.

Instances of FMonadT should make tmap respect the identity natural transformation and composition of natural transformations. Instances of MMonadT should make flift be equal to lift (although this seems to make flift superfluous, it has different type-class requirements as lift requires a Monad, while flift requires a Functor), monoidalT should be associative, and flift at the identity functor (this corresponds to ϕ_I in Definition 2.5) should be a left and right unit for monoidalT.

```
class MonadT t ⇒ FMonadT t where
  tmap :: (Functor m, Functor n) ⇒ (∀ b. m b → n b) → t m a → t n a
class FMonadT t ⇒ MMonadT t where
  flift      :: Functor f ⇒ f a → t f a
  monoidalT :: (Functor f, Functor g) ⇒ t f (t g a) → t (f ∘ g) a
```

where the type $\circ$ expresses functor composition (see Appendix A.6).

The codensity monad transformer and its down operation are defined as follows:

```
newtype Cod f a = Cod {unCod :: ∀ b. (a → f b) → f b}
down   :: Monad m ⇒ Cod m a → m a
down c = unCod c return
instance MonadT Cod where
  tbind c f = Cod (λk → unCod c (λa → unCod (f a) k))
  lift m    = Cod (m >>=)
instance Monad m ⇒ Monad (Cod m) where
  return = treturn
  (>>=)  = tbind
instance Functor f ⇒ Functor (Cod f) where
  fmap f (Cod c) = Cod (λk → c (k · f))
```

The implementation of a particular operation for a monad provides a *model* of the operation. We define three types of models and for each of these we provide a generic lifting

function. This corresponds to each of the three classes of operations, transformers and liftings in Figure 4.1. Each of the lifting functions is constrained to the corresponding monad transformer class and type of operation.

```
type AlgModel f m = ∀ a. (Functor f, Monad m) ⇒ f a → m a
liftAlgModel      :: (MonadT t, Monad m, Functor f) ⇒ AlgModel f m → AlgModel f (t m)
liftAlgModel mdl = lift · mdl
```

```
type Model f m = ∀ a. (Functor f, Monad m) ⇒ f (m a) → m a
liftModel      :: (Functor f, Monad m, Functor m, FMonadT t, Monad (t (Cod m))) ⇒
                  Model f m → Model f (t m)
liftModel mdl = tmap down · join · lift · toAlg mdl · fmap (tmap lift)

toAlg          :: (Functor f, Monad m) ⇒ Model f m → AlgModel f (Cod m)
toAlg mdl t    = Cod (λk → mdl (fmap k t))
```

```
type ExtModel f g m = ∀ a.f (m (g a)) → m a
liftExtModel      :: (Functor f, Functor g, Monad m, Functor m,
                      MMonadT t, Functor (t f), Functor (t m)) ⇒
                     ExtModel f g m → ExtModel f g (t m)
liftExtModel mdl = tmap (mdl · fmap deComp · deComp) · monoidalT · flift ·
                      fmap (monoidalT · fmap flift)
```

The function join in the definition of liftModel is the multiplication of a monad, and it is defined as follows.

```
join :: Monad m ⇒ m (m a) → m a
join = (≫=id)
```

Operations

For each (set of) operations we define a signature functor and functions associated to that signature functor which given a model perform an effect-manipulating operation (identified by the X suffix, and called X-*operations* henceforth). For example, the signature for side-effect operations and its associated X-operations are:

```
data StateOp s a = Get (s → a) | Put s a
instance Functor (StateOp s) where
  fmap f (Get g)   = Get (f.g)
  fmap f (Put s a) = Put s (f a)

getX       :: Monad m ⇒ AlgModel (StateOp s) m → m s
getX mdl   = mdl (Get id)
```

```
putX      :: Monad m ⇒ AlgModel (StateOp s) m → s → m ()
putX mdl s = mdl (Put s ())
```

The X-operations do not need to have the same type as the one dictated by the signature (for example, $(s \to a) \to m\ a$ for accessing the state) but can be presented in a form which is more familiar to programmers, as in the example above.

As another example, we present the operations for throwing and handling exceptions. In this case one of the operations is algebraic but the other is not. We choose to separate the operations into two separate signatures, so that it is possible to lift the operation Throw through arbitrary monad transformers.

```
data ThrowOp x a = Throw x
instance Functor (ThrowOp x) where
  fmap _ (Throw x) = Throw x
throwX      :: Monad m ⇒ AlgModel (ThrowOp x) m → x → m a
throwX mdl x = mdl (Throw x)
```

```
data HandleOp x a = Handle a (x → a)
instance Functor (HandleOp x) where
  fmap f (Handle a h) = Handle (f a) (f . h)
handleX      :: Monad m ⇒ Model (HandleOp x) m → m a → (x → m a) → m a
handleX mdl m h = mdl (Handle m h)
```

With this interface the problem of shadowing disappears. It is now possible, when using an operation, to state explicitly and precisely which model is meant and through how many monad transformers this model should be lifted. For example, if modelHandleExcT is the model of handle as provided by the exception monad transformer, then the operation handleX modelHandleExcT is the operation for handling exceptions as implemented by modelHandleExcT. Its lifting through two functorial monad transformers is given by handleX (liftModel (liftModel modelHandleExcT)).

Overloading of operations

It is simple to add overloading on top of the core functionality. Define a type class of monads with models of a given signature, and then show that all the appropriate monad transformers for that kind of models are also in the same type class (i.e. they also have model). Finally, define operations which work on any monad of the appropriate type class.

In the following we present the case of Models. The other two cases (AlgModel and ExtModel) are analogous. We define the class of monads m which have a model of signature f:

```
class (Functor f, Monad m) ⇒ MonadModel f m where
  model :: Model f m
instance (..., FMonadT t, MonadModel f m) ⇒ MonadModel f (t m) where
  model = liftModel model
```

The instance—where we omit a part of the class context—indicates that if a monad m has a model Model f m, then also t m has a model (when t is a functorial monad transformer). In order to obtain operations with automatic liftings, we replace the explicit parameter in handleX (a model Model (HandleOp x) m) by an implicit parameter given by the type class.

```
handle :: MonadModel (HandleOp x) m ⇒ m a → (x → m a) → m a
handle = handleX model
```

Finally, we provide instances for the concrete monads that implement the operations: in this case, the monad ExcT x m (for any monad m).

```
instance Monad m ⇒ MonadModel (HandleOp x) (ExcT x m) where
  model = modelHandleExcT
```

Of course, implementing overloading of operations will require the use of many extensions to Haskell 98, as it was the case with the traditional design of monad transformer libraries. However, in this case, the core functionality does not rely on the language extensions and one has the choice of using the extensions when they are available and overloading is convenient, as opposed to being forced to always do so.

Implementing Transformers and Models

So far, we have defined functions that deal with generic implementations of certain operations. For example, handleX deals with models Model (HandleOp x) m. We now define concrete monad transformers and show that they provide a model of its associated operations. In particular, we will define the side-effect monad transformer and its model of StateOp operations, and the exception monad transformer and its models of ThrowOp and HandleOp.

Side-Effect Monad Transformer

We start with the side-effect monad transformer.

```
newtype StateT s m a = S {unS :: s → m (a, s)}
runStateT :: s → StateT s m a → m (a, s)
runStateT s m = unS m s
instance MonadT (StateT s) where
  tbind m k = S (λs → unS m s ≫= λ(a, s') → unS (k a) s')
  lift m    = S (λs → m ≫= λa → return (a, s))
```

```
instance FMonadT (StateT s) where
  tmap f (S m) = S (f · m)
instance MMonadT (StateT s) where
  flift t = S (λs → fmap (λa → (a, s)) t)
  monoidalT (S t) = S (λs → Comp (fmap (λ(S t', s') → t' s') (t s)))
instance Monad m ⇒ Monad (StateT s m) where
  return = treturn
  (>>=) = tbind
```

The monad transformer StateT provides a model of StateOp:

```
modelStateT           :: Monad m ⇒ AlgModel (StateOp s) (StateT s m)
modelStateT (Get g)   = S (λs → return (g s, s))
modelStateT (Put s a) = S (λ_ → return (a, s))
```

Once we have defined the model, we can use it via getX and putX, and lift it with liftAlgModel. Importantly, there is a separation of the notion of *model* of operations (as provided by modelStateT) and the *use* of that type of operation (as provided by the functions getX, putX, and liftAlgModel).

Exception Monad Transformer

The exception monad transformer ExcT is defined as follows:

```
newtype ExcT x m a = X { unX :: m (Either x a) }
runExcT :: ExcT x m a → m (Either x a)
runExcT = unX
instance MonadT (ExcT x) where
  tbind (X m) f = X (do a ← m
                        case a of Left x → return (Left x)
                                  Right b → unX (f b))
  lift m        = X (liftM Right m)
instance FMonadT (ExcT x) where
  tmap f = X · f · unX
instance Monad m ⇒ Monad (ExcT x m) where
  return = treturn
  (>>=) = tbind
```

It provides, for any monad m, an algebraic model AlgModel (ThrowOp x) (X x m), and a model Model (HandleOp x) (X x m).

```
modelThrowExcT           :: Monad m ⇒ AlgModel (ThrowOp x) (ExcT x m)
modelThrowExcT (Throw x) = X (return (Left x))
```

```
modelHandleExcT                  :: Monad m ⇒ Model (HandleOp x) (ExcT x m)
modelHandleExcT (Handle m h) = X (unX m ≫= λexa → case exa of
                                                    Left x  → unX (h x)
                                                    Right a → return (Right a))
```

More examples of monad transformers and their operations can be found in Appendix A, where the full source of the Monatron library is provided.

5.4 Summary

Combinator libraries for monads are essential for facilitating the construction of complex monads that naturally appear in applications that go from basic parser libraries (Hutton and Meijer 1998) to end-user applications (Stewart and Sjanssen 2007). We have shown that the current design of monad transformer libraries has a number of shortcomings that hinder the extensibility, predictability, portability, and expressive power of the library.

By restructuring the design and incorporating uniform liftings of operations we have managed to address these issues. The approach has several advantages:

Uniform-liftings: operations are lifted uniformly through monad transformers by construction. This means that the semantics of the lifted operations is predictable.

Modularity: operations need to be defined only once for each monad/monad transformer that supports them, effectively reducing the quadratic growth of number of instances to linear.

Expressivity: One has the ability to exactly state which operation one is referring to. In fact, if desired, one can have more that one model of an operation for a given monad, since all operations and liftings are parametrised by arbitrary models.

The implementation that follows the ideas above constitutes the *core* of our library Monatron. This core only needs Haskell 98 (Peyton Jones 2003) extended with rank-2 types (Peyton Jones et al. 2007) and provides full functionality, but no overloading of operations. However, the overloading of operations provided by type classes is often convenient. When there is no possible shadowing, and using additional language extensions is not problematic, one can let the compiler infer to which model one is referring to.

The operations in the library must be constrained to one of the three formats provided. This means that a programmer extending the library with a new monad transformer has to be careful about how to define the effect-manipulating operations, for example analyzing if the operations is algebraic or not, in order to get the most general lifting. Additionally, the new transformer need to be studied to see if it is monoidal or functorial. However, once

this is done, there is no need to do any additional work in order to lift the new operations through existing monad transformers, or to lift existing operations through the new monad transformer, as this is taken care of by the library infrastructure.

One of the design decisions was to maintain hidden the implementation of transformers and operations. This may prove to be useful for improving the efficiency of the library, as hidden implementations allow for optimizations that preserve the interface. For example, the list monad is often used for modelling non-determinism, but its *merge* operation (concatenation) is rather inefficient. Using a different internal structure, but preserving the interface, we can provide an efficient *merge* operation. We leave as future work a further departure from the traditional implementation of monads in search for better performance.

The mtl (*Monad Transformer Library*) is the most well-known monad transformer library. It is inspired by the work of Liang, Hudak, and Jones (1995), and for many years it has been distributed together with the Haskell compiler GHC. More recently, a new library called MonadLib (*MonadLib*) has been introduced. This library is an improvement over the mtl, but it still suffers from the problems described in Section 5.2. However, the library presented in this article owes a lot to the excellent work done by the authors of these two libraries.

The codensity monad transformer has appeared in a number of functional programming papers. For example, it has been derived as a monad transformer for backtracking (Hinze 2000), it has been calculated in a search for efficient parsers (Claessen 2004), and it has been used to optimize substitution in the free monad (Voigtländer 2008). In our case, however, we were motivated by its mathematical properties.

Chapter 6

Modular Interpreters Revisited

In this chapter we implement a modular interpreter for an example language. The language is constructed by combining a process algebra, an arithmetic language, and a language for exceptions. The language is later extended with parallel processes.

The purpose of the implementation is two-fold.

- The modular interpreter serves as an extended example of the use of monad transformers in general, and of programming using Monatron, in particular.
- The example language will be reused in the next chapter on modular operational semantics. This will allow us to compare the similarities, strengths and limitations of the two approaches.

6.1 Modular Syntax

The first step towards obtaining modular semantics is to obtain modular syntax, in the sense that the terms of a language are constructed by combining smaller languages. Consider a simple process language P whose terms $p \in P$ are specified by the following grammar, which corresponds to Basic Process Algebra (Bergstra and Klop 1985; Fokkink 2000) with empty process (Bergstra, Fokkink, and Ponse 2001):

$$p \ ::= \ \mathsf{nil} \ \mid \ !_a \ \mid \ p\,;p \ \mid \ p \sqcup p$$

where a is a character. The informal meaning of the operators in the language is that $!_a$ performs an atomic action a, which for the purposes of this chapter can be thought of as the printing of the character a on the screen, $p\,;q$ sequences the execution of p and q, and $p \sqcup q$ non-deterministically chooses to execute either p or q.

It is straightforward to implement the grammar for P as a recursive datatype:

```
data P = Nil | Put Char | Seq P P | Alt P P
```

However, datatype P is monolithic. In order to obtain modular syntax we need to reveal the underlying structure, separating the operators of the language from the description of its terms. We use the standard categorical technique (for example, see Lüth and Ghani 2002) of modelling terms by the free monad over a signature, and the combination of languages by the coproduct of free monads.

Terms as Free Monads

We will specify the syntax of a language by its *signature*, that is, the set of its operators and their corresponding arities. Each signature has a corresponding instance of the Functor class, which is its *signature functor*.

Example 6.1. The signature functor for P is as follows:

```
data P a = Nil | Put Char | Seq a a | Alt a a

instance Functor P where
  fmap _ Nil       = Nil
  fmap _ (Put c)   = Put c
  fmap f (Seq p q) = Seq (f p) (f q)
  fmap f (Alt p q) = Alt (f p) (f q)
```

Example 6.2. We define a simple language of arithmetic expressions, with integers, additions and a conditional expression, whose terms $z \in Z$ are specified by the following grammar:

$$z \ ::= \mathbb{Z} \mid z + z \mid \mathsf{ifz}\ z\ z\ z$$

The informal meaning of ifz c t e is that if c is 0 then t is evaluated, otherwise e is evaluated.

The signature functor for Z is as follows:

```
data Z a = Num Int | Add a a | Ifz a a a

instance Functor Z where
  fmap f (Num i)   = Num i
  fmap f (Add p q) = Add (f p) (f q)
  fmap f (Ifz c t e) = Ifz  (f c) (f t) (f e)
```

Example 6.3. Let us consider now a language E of exceptions:

$$e \ ::= \ \mathsf{throw} \mid \mathsf{catch}\ e\ e$$

The informal meaning is that throw throws an exception and catch t u evaluates t and, if t throws an exception, it recovers from it by evaluating u.

The signature functor for E is as follows:

```
data E a = Thr | Cat a a
```

```
instance Functor E where
  fmap _ Thr       = Thr
  fmap f (Cat p q) = Cat (f p) (f q)
```

This language is not very useful by itself as the only possible outcome is to throw an exception. Its real utility is exhibited when one considers the language E together with some other language.

Terms constructed with operators from the signature functor f and with variables of type x are given by the free monad on f at x, represented by the datatype $\mathsf{Term}\ f\ x$. The flexibility of having variables of an arbitrary type will play a significant role in the next chapter where they are used to represent meta-variables in operational rules.

```
data Term f x = Var x | Con (f (Term f x))
```

That is, a term is either a variable or an operator from f applied to a term. It is now straightforward to make such terms into both Functors and Monads:

```
instance Functor f ⇒ Functor (Term f) where
  fmap f (Var x)   = Var (f x)
  fmap f (Con t)   = Con (fmap (fmap f) t)
instance Functor f ⇒ Monad  (Term f) where
  return           = Var
  (Var x) >>= f    = f x
  (Con t) >>= f    = Con (fmap (>>=f) t)
```

The fact that $\mathsf{Term}\ f$ is a monad shows that terms structured in this way come equipped with a substitution operator, as given by $(\gg\!=) :: \mathsf{Term}\ f\ a \to (a \to \mathsf{Term}\ f\ b) \to \mathsf{Term}\ f\ b$ (Ghani and Lüth 1997; Mac Lane 1971). With this representation of terms, the natural manner in which to process terms is using a generic fold operator (Hagino 1987; Meijer, Fokkinga, and Paterson 1991):

```
foldTerm :: (Functor f) ⇒ (a → b) → (f b → b) → Term f a → b
foldTerm v _ (Var a)   = v a
foldTerm v c (Con fta) = c (fmap (foldTerm v c) fta)
```

Intuitively, the argument of type $a \to b$ is used to process variables, and the argument of type $f\ b \to b$ (an f-algebra) is used to process operators.

Finally, the *programs* of a language are its closed terms. That is, programs are terms with variables taken from the empty datatype Zero, which comes equipped with a canonical map $\mathsf{empty} :: \mathsf{Zero} \to a$ into any other type a.

```
type Program f = Term f Zero
```

Thus, we have a generic notion of syntax equipped with well-behaved substitution and a well-behaved recursion operator. Moreover, as shown in the next section, we obtain a simple and principled method for combining the syntax of languages.

Coproducts of Free Monads

We have shown that signatures define the operators of a language. The natural way to combine two languages is to take the coproduct of the free monads modelling them. Since free constructions preserve coproducts, this is equivalent to the free monad on the coproduct of their signature functors. Consequently, we define the coproduct of functors as follows.

```
data (f ⊕ g) a      = Inl (f a) | Inr (g a)
instance (Functor f, Functor g) ⇒ Functor (f ⊕ g) where
  fmap h (Inl fx)  = Inl (fmap h fx)
  fmap h (Inr gx) = Inr (fmap h gx)
copair              :: (f a → b) → (g a → b) → (f ⊕ g) a → b
copair f _ (Inl fa)  = f fa
copair _ g (Inr ga) = g ga
```

The function copair processes the coproduct of functors f and g, given that we provide two functions: one to process f and the other to process g. We can use copair to define an $f \oplus g$-algebra from an f-algebra and a g-algebra. Therefore, foldTerm can be used to process $f \oplus g$ terms.

Example 6.4. We can combine the signature of the languages P, Z, and E using coproducts:

```
type L = P ⊕ Z ⊕ E
```

The term of the combined language L $(\,!_a\,;(3+5)) \sqcup (\mathit{catch}\ \mathit{throw}\ !_c)$ is written in Haskell as the program prog:

```
prog :: Program L
prog = (seq (put 'a') ((n 3) + (n 5))) ⊔ (catch throw (put 'c'))
put                 :: Char → Term L a
put c               = Con (Inl (Put c))
n                   :: Int → Term L a
n m                 = Con (Inr (Inl (Num m)))
throw               :: Term L a
throw               = Con (Inr (Inr Thr))
seq, · ⊔ ·, · + ·, catch :: Term L a → Term L a → Term L a
seq p q             = Con (Inl (Seq p q))
p ⊔ q               = Con (Inl (Alt p q))
```

$$
\begin{aligned}
p + q \quad & = \mathsf{Con}\ (\mathsf{Inr}\ (\mathsf{Inl}\ (\mathsf{Add}\ p\ q))) \\
\mathsf{catch}\ p\ q \quad & = \mathsf{Con}\ (\mathsf{Inr}\ (\mathsf{Inr}\ (\mathsf{Cat}\ p\ q)))
\end{aligned}
$$

Note that we may define other languages by taking other combinations. For example, we may define the syntax EP of basic process algebra extended with exceptions or define EZ of arithmetic extended with exceptions.

$$
\begin{aligned}
&\textbf{type}\ \mathsf{EP} = \mathsf{P} \oplus \mathsf{E} \\
&\textbf{type}\ \mathsf{EZ} = \mathsf{Z} \oplus \mathsf{E}
\end{aligned}
$$

Coproducts provide a structured, mathematical foundation for assembling syntax. There are, however, some practical concerns. As shown in example 6.4, we had to define auxiliary functions such as put, seq, and $\cdot \sqcup \cdot$ in order to make the definition of programs less cumbersome. These shorthands will only work for terms of signature P, and would need to be changed should the language be extended. For instance, if we were working with the language EP, then we would have to define a new auxiliary function throw' as:

$$
\begin{aligned}
&\mathsf{throw}' :: \mathsf{Term}\ \mathsf{EP}\ a \\
&\mathsf{throw}' = \mathsf{Con}\ (\mathsf{Inr}\ \mathsf{Thr})
\end{aligned}
$$

Redefining these auxiliary functions every time we change our language is inherently non-modular. In the following subsection we show how to solve this problem.

Automatic Injections and Partial Projections

Let $\Sigma = F_1 \oplus \ldots \oplus F_n$ be a coproduct of functors such that $i \neq j \Rightarrow F_i \neq F_j$. Then, if $G = F_i$ we can talk about an injection in_G rather than in_i, hence avoiding the need to explicitly state the index of G in the coproduct. In this situation, it is possible to define injections and do case analysis of coproducts for which only a particular type is known (in which case we call the case analysis a *partial projection*).

The way to achieve this (Liang, Hudak, and Jones 1995; Swierstra 2008) is to parameterise each function by injection/projection pairs corresponding to each of the summands a producer/consumer of the coproduct is interested in. Rather than explicitly parameterising each function, we use Haskell's type-class system and let the compiler figure out which injection/projection is meant.

$$
\begin{aligned}
&\textbf{class}\ \mathit{sub} \hookrightarrow \mathit{sup}\ \textbf{where} \\
&\quad \mathsf{inj} :: \mathit{sub}\ a \to \mathit{sup}\ a \\
&\quad \mathsf{prj} :: \mathit{sup}\ a \to \mathsf{Maybe}\ (\mathit{sub}\ a)
\end{aligned}
$$

We can think of $\mathit{sub} \hookrightarrow \mathit{sup}$ as meaning "*sub* is a subtype of *sup*". The class method inj is used to inject a subtype *sub* into the supertype *sup*, and prj let us do a case analysis on a *sup* to determine if it is in fact a *sub*.

The following instances state the reflexivity of $\cdot \hookrightarrow \cdot$ and that f is a subtype of a coproduct $g_1 + (g_2 + (\ldots))$ when either $f = g_1$ or f is a subtype of $(g_2 + (\ldots))$. Importantly, the sum must be associated to the right for the type-checker to be able to infer an instance, so we need to be careful when constructing coproducts.

```
instance f ↪ f where
  inj = id
  prj = Just

instance f ↪ f ⊕ g where
  inj          = Inl
  prj (Inl f)  = Just f
  prj _        = Nothing

instance (f ↪ g) ⇒ f ↪ h ⊕ g where
  inj          = Inr · inj
  prj (Inr a)  = prj a
  prj _        = Nothing
```

Finally, we will rewrite the auxiliary functions in Example 6.4 so that they work with any signature which satisfies certain requirements expressed as type constraints. In Figure 6.1 we show the modular constructors for the operators in P, Z, and E.

The type-class $\hookrightarrow$ provides injections and case analysis for types of kind $* \to *$. We apply the same technique for types of kind $*$ and define the following type-class and instances.

```
class sub ↪* sup where
  inj* :: sub → sup
  prj* :: sup → Maybe sub

instance v ↪* v where
  inj* = id
  prj* = Just

instance v ↪* Either v u where
  inj*           = Left
  prj* (Left v)  = Just v
  prj* _         = Nothing

instance (v ↪* w) ⇒ v ↪* Either u w where
  inj*           = Right · inj*
  prj* (Right a) = prj* a
  prj* _         = Nothing
```

The instances for $\overset{*}{\hookrightarrow}$ are analogous to the instances for $\hookrightarrow$ except that instead of the coproduct of functors $\oplus$ we use the datatype Either. For convenience, we define a function

```
con      :: (s ↪ t) ⇒ s (Term t x) → Term t x
con      = Con · inj
nil      :: (P ↪ t) ⇒ Term t x
nil      = con Nil
put      :: (P ↪ s) ⇒ Char → Term s x
put c    = con (Put c)
seq      :: (P ↪ s) ⇒ Term s x → Term s x → Term s x
seq p q  = con (Seq p q)
· ⊔ ·    :: (P ↪ s) ⇒ Term s x → Term s x → Term s x
p ⊔ q    = con (Alt p q)
n        :: (Z ↪ s) ⇒ Int → Term s x
n m      = con (Num m)
· + ·    :: (Z ↪ s) ⇒ Term s x → Term s x → Term s x
p + q    = con (Add p q)
ifz      :: (Z ↪ s) ⇒ Term s x → (Term s x, Term s x) → Term s x
ifz c (t,e) = con (Ifz c t e)
thr      :: (E ↪ s) ⇒ Term s x
thr      = con Thr
cat      :: (E ↪ s) ⇒ Term s x → Term s x → Term s x
cat p q  = con (Cat p q)
```

Figure 6.1: Modular constructors for the operators of P, Z, and E

$\gg\!=^{\hookrightarrow}$ which acts like $\gg\!=$, but works on monads on a supertype, and only binds a subtype.

```
(>>=↪)  :: (sub ↪* sup, Monad m) ⇒ m sup → (sub → m sup) → m sup
m >>=↪ f = do x ← m
              case prj* x of
                   Nothing → return x
                   Just y  → f y
```

With the use of coproducts and the functorial representation of signatures, we achieved our goal of obtaining and implementing a modular syntax.

6.2 Modular Interpreters

We now present how to combine the semantics of each language into the semantics for the total language. Since an interpreter for a language with signature functor f is given by an f-algebra, we define the class Interp of functors with an algebra over a given computational monad M and type of values V. Evaluating a program of a language f which is an instance of Interp is simply folding the algebra interp.

```
class (Functor f) ⇒ Interp f where
  interp :: f (M V) → M V
eval :: (Interp f) ⇒ Program f → M V
eval = foldTerm empty interp
```

Given two languages f and g which are instances of Interp, the semantics of the combined language is given by the copair of the corresponding algebras.

```
instance (Interp f, Interp g) ⇒ Interp (f ⊕ g) where
  interp = copair interp interp
```

Computations and Values

The class Interp is defined over a computational monad M and value type V For defining L we need a computational monad that can do exceptions, output, non-determinism, and a value type that can express the null process and integers. We take the monad M and type V of values to be:

```
type V = Either Int ()
type M = ExcT () (WriterT String (NonDetT Id))
```

The value type V is defined as either an Int (as produced by the arithmetic language) or unit (as produced by the process algebra language). The type V could have been defined as Maybe Int, but by using Either we benefit from the automatic injections and partial projections provided by the type class $\overset{*}{\hookrightarrow}$.

The monad M is constructed by successively applying monad transformers to the monad of pure computations Id. The applied monad transformers are:

- the NonDetT monad transformer to add non-determinism;
- the WriterT String monad transformer to add string output;
- the ExcT () monad transformer to add exceptions of type unit.

Full details of the implementation of these three monad transformers are available in Appendix A.

6.3 Interpreters for the Sub-Languages

We define the interpreters of the languages that are assembled to form L by providing instances of the type class Interp for each signature functor. In each case, the interpreters are defined assuming as little as possible about the monad M and the type of values V.

Interpreter for Z

The interpreter for the arithmetic language Z is given by the following instance.

```
instance Interp Z where
  interp (Num n)   = return (inj* n)
  interp (Add x y) = x >>=↪ λm →
                     y >>=↪ λn →
                     return (inj* (n + m :: Int))
  interp (Ifz c t e) = c >>=↪ λm → if (m :: Int) ≡ 0 then t else e
```

The use of $\gg\!=^{\hookrightarrow}$ allows us to define this instance of Interp with only the knowledge that Int is a subtype of V. We had to add type annotations so that the compiler can infer the type of the injection inj_*, as addition an 0 are overloaded in Haskell and they can refer to any type which is an instance of the type-class Num. There no requirements on M apart from it being a monad.

Interpreter for E

The interpreter for the exception language E is given by the following instance. In this case the semantics is discharged on the effectful operations of the monad M.

```
instance Interp E where
  interp Thr       = throw ()
  interp (Cat a h) = handle a (λ() → h)
```

The interpreter for E does not need any knowledge of the values in V, but requires a monad M which supports the operations throw and handle.

Interpreter for P

The interpreter for the process algebra language P is given by the following instance.

```
instance Interp P where
  interp Nil       = return (inj* ())
  interp (Put c)   = trace [c] >>= return · inj*
  interp (Seq t u) = t >>= λ_ → u
  interp (Alt t u) = plusND t u
```

The interpreter for P requires a monad M which supports operations for writing String traces and for non-determinism. The semantics of Put requires that the unit type () is a subtype of V.

We have defined instances of Interp providing interpreters for the languages Z, E, and P. The instance for the combined language L is automatically obtained by the previously defined instance of Interp for coproducts.

6.4 Adding Parallel Computations

If we want to extend our language with a *merge* operator for adding parallel computation of processes we need to:

- Define the syntax for the operator (and provide a modular constructor)

  $$\textbf{data}\ \mathsf{R}\ a = \mathsf{Par}\ a\ a$$

  $$\textbf{instance}\ \mathsf{Functor}\ \mathsf{R}\ \textbf{where}$$
  $$\quad \mathsf{fmap}\ f\ (\mathsf{Par}\ p\ q) = \mathsf{Par}\ (f\ p)\ (f\ q)$$
  $$\cdot \parallel \cdot :: (\mathsf{R} \hookrightarrow s) \Rightarrow \mathsf{Term}\ s\ x \to \mathsf{Term}\ s\ x \to \mathsf{Term}\ s\ x$$
  $$p \parallel q = \mathsf{con}\ (\mathsf{Par}\ p\ q)$$

- Extend the computational monad M in a way that allows us to define the desired semantics. The intuitive understanding of the merge operator is that two processes are run in parallel, interleaving atomic actions or steps. For this we use the step monad transformer (for details, see Appendix A.1).

  $$\textbf{codata}\ \mathsf{StepT}\ f\ m\ x = \mathsf{T}\ \{\mathsf{runT} :: m\ (\mathsf{Either}\ x\ (f\ (\mathsf{StepT}\ f\ m\ x)))\}$$

  In Haskell there is no distinction between least and greatest fixpoints of recursive datatypes. To distinguish between them, we write least fixpoints as **data** and greatest fixpoints, such as StepT as **codata**. Note that the related monad transformer of example 4.6 is a least fixpoint and can only represent the interleaving of terminating processes.

The monad $\mathsf{StepT}\ f\ m$ has steps of type f and supports an operation $\mathsf{step} :: f\ (m\ a) \to m\ a$ (see Appendix A.4). The intended notion of step for our language is to print a character. That is, we will consider printing a character (possibly with other effects) as an atomic action. Therefore, we define the functor Pr for representing this notion of step and we show that for any monad m, the monad $\mathsf{StepT}\ \mathsf{Pr}\ m$ has the algebraic operation trace:

$$\textbf{data}\ \mathsf{Pr}\ a = \mathsf{Pr}\ \mathsf{Char}\ a$$

$$\textbf{instance}\ \mathsf{Functor}\ \mathsf{Pr}\ \textbf{where}$$
$$\quad \mathsf{fmap}\ f\ (\mathsf{Pr}\ c\ a) = \mathsf{Pr}\ c\ (f\ a)$$

$$\textbf{instance}\ (\mathsf{Monad}\ m) \Rightarrow \mathsf{MonadAlgModel}\ (\mathsf{WriterOp}\ \mathsf{Char})\ (\mathsf{StepT}\ \mathsf{Pr}\ m)\ \textbf{where}$$
$$\quad \mathsf{algModel}\ (\mathsf{Trace}\ c\ a) = \mathsf{step}\ (\mathsf{Pr}\ c\ (\mathsf{return}\ a))$$

We redefine the monad M, replacing the WriterT String monad transformer by the monad transformer for Pr-steps. Importantly, there is no need to modify existing interpreters, as this new combined monad supports all the operations required by them.

```
type M = ExcT () (StepT Pr NonDet)
```

Finally, we provide the interpreter for the parallel merge operation:

```
instance Interp R where
  interp (Par t u) = t ⊗ u
    where t ⊗ u = plusND (caseStep^ExcT (const u)
                                        (λft → step (fmap (⊗u) ft)) t)
                         (caseStep^ExcT (λv → liftM (const v) t)
                                        (λfu → step (fmap (t⊗) fu)) u)
```

where the function

```
liftM      :: (Monad m) ⇒ (a → b) → m a → m b
liftM f m = m >>= return · f
```

maps a function under a monad, in the same way that fmap maps a function under a functor.

The parallel merge is defined by non-deterministically doing a step on the left argument or on the right argument. This implementation of parallel execution is not symmetric. The term on the left is only executed for side-effects and its resulting value is discarded. This kind of parallelism is commonly seen in functional languages where one is interested in both the effects and the value of an expression (for another example of this style of semantics, see Peyton Jones, Gordon, and Finne 1996).

In order to define the parallel merge operation, we needed to use the caseStep operation of the StepT monad transformer.

```
caseStep          :: (Functor f, Monad m) ⇒
                     (a → StepT f m x) → (f (StepT f m a) → StepT f m x)
                     → StepT f m a → StepT f m x
caseStep v c (T m) = T (m >>= either (runT · v) (runT · c))
```

This operation does not fit in any of our formats of liftable operations (see the discussion at the end of Section 3.2). Consequently, we have to manually lift it trough the exception monad transformer.

```
caseStep^ExcT :: (Functor f, Monad m) ⇒
                 (a → ExcT e (StepT f m) x) →
                 (f (ExcT e (StepT f m) a) → ExcT e (StepT f m) x) →
                 ExcT e (StepT f m) a → ExcT e (StepT f m) x
caseStep^ExcT v c = X · caseStep (runExcT · either throw v) (runExcT · c · fmap X) · runExcT
```

6.5 Summary

We have seen how monad transformers and the Monatron library can be used to program modular interpreters. In particular, we combined three languages, and then extended the resulting language with a parallel merge operator without the need to modify the existing interpreters.

We have shown how to combine syntax modularly. The technique is based on the simple fact that, since left adjoints preserve colimits, the coproduct of two free monads $T_F + T_G$ is equivalent to T_{F+G} (i.e. the free monad on the coproduct $F + G$). The implementation of the type class that provides automatic injections and partial projections for types of kind $*$ is originally from Liang, Hudak, and Jones (1995), and has been implemented for kind $* \to *$ and explained in detail by Swierstra (2008).

The combination of the languages for exceptions, arithmetic and basic process algebra worked seamlessly: The interpreters for each of these languages were defined independently, with each language posing certain requirements on the computational monad and type of values. It is interesting to note that, because the order in which transformers are applied to construct a monad matters, we can define many different semantics by instantiating M to different monads.

The addition of a construct for parallelism required modifying the existing monad for a more refined version which incorporated a notion of "step". In the literature the resumptions monad transformer is often used for this purpose (see, for example, Espinosa 1994). The resumptions monad transformer is equivalent to the step monad transformer on a trivial step (i.e. where the step functor is the identity functor). In other papers (for example, Papaspyrou 2001)[1] the completely iterative monad (Milius 2005) on the underlying functor of the transformed monad is presented as the resumptions monad transformer. However this construction does not fit the standard notion of monad transformer as it does not have a monad morphism lift. When using the resumptions monad, interleaving points have to be manually inserted in the semantics, for example, after printing a character. We prefer the use of the StepT monad transformer as it does not require us to modify the existing semantics.

One problem with the use of the StepT monad transformer is that we were required to lift caseStep manually (this problem would also occur had we used the resumptions monad transformer). A more satisfying solution for lifting operations such as caseStep is clearly needed.

The idea of a type-class Interp for signatures with an interpreter instance, is inspired by the modular interpreter implementation of Liang, Hudak, and Jones (1995).

[1] In these papers the ambient category is assumed to be algebraically compact and, consequently, they make no distinction between least and greatest fixpoints.

Chapter 7

Modular Operational Semantics

Operational semantics is one of the primary techniques for formally specifying the meaning of programs. Traditionally, one defines the operational semantics of a programming language as a relation over the syntax of programs. In structural operational semantics (Plotkin 1981; reprinted in Plotkin 2004), this relation is defined by a set of inductive rules. Moreover, one seeks syntactic restrictions of the format of rules in order to guarantee certain properties. The simplicity of this approach has made structural operational semantics very popular, especially for concurrent languages. Nevertheless, the syntactic nature of this approach to operational semantics means that it is difficult to establish language-independent, meta-theoretical results. In the absence of a non-syntactic meta-theory one is faced with the following problems:

- It is not clear how the syntactic restrictions on rules that are needed to obtain a sensible notion of equivalence arise, or how they can be modified to accommodate changes in the language or in the notion of observable behaviour. The lack of an abstract meta-theory means that different rule formats have to be developed independently in order to accommodate different language features (for an overview of different rule formats and language features supported, see Aceto, Fokkink, and Verhoef 2001).

- It is not clear how to relate operational semantics with denotational semantics in a language-independent manner. One would like to reason about programs with the more abstract denotational semantics, for which general, language-independent tools are available, and use the operational semantics to understand how programs would be executed in a machine. However, without a meta-theory which relates the two approaches, proofs of adequacy need to be done for each language.

- Without an abstract meta-theory it is difficult to express a generic notion of operational semantics in a programming language or in a theorem prover. Without a way to express such a generic notion, it is hard to imagine how one could develop a framework

for operational semantics similar to those existing for implementing logics or type theories.

Fortunately, the development of an abstract meta-theory for operational semantics has begun. Turi (1996) abstracted from the concrete, syntactic approach to operational semantics, and expressed operational semantics as a categorical construct, namely a distributive law of syntax over behaviour. By parameterising his construct by a functor representing syntax and a functor representing semantics, Turi abstracted away from the specific details of particular languages and their meaning. Moreover, it became possible to relate the operational and denotational approaches; indeed, they become two sides of the same coin, as they define the same semantic function, one by the universal property of the final coalgebra, the other by the universal property of the initial algebra. That is, the semantic function $[\![-]\!]\colon \mu\Sigma \to \nu B$ from the syntax into the behaviour, is induced by both an algebra over the final coalgebra νB and a coalgebra over the initial algebra $\mu\Sigma$. These are provably equal.

In previous chapters we have shown how to modularly combine denotational semantics using monads and monad transformers. In this chapter, we take advantage of the relation between the operational and the denotational approaches exposed by Turi, and use monads and monad transformers to obtain modular operational semantics.

We implement our ideas in Haskell, which helps to bring Turi's categorical work to the functional programming community in a more accessible way, makes the ideas directly executable, facilitates experimentation, and allows us to benefit from Haskell's well-developed support for monadic programming. Moreover, it will make more direct the comparison between the obtained modular operational semantics and the modular interpreters of the previous chapter.

7.1 Structural Operational Semantics

Operational semantics gives meaning to terms in a language by defining a transition relation that captures execution steps in an abstract machine. Reasoning about this relation can be difficult. Therefore Plotkin proposed *structural operational semantics* (SOS), in which the transition relation is defined by structural recursion on syntax-directed rules (Plotkin 1981, 2004). One then uses the principle of structural induction to reason about the induced transition relation.

We give as examples the structural operational semantics for the languages P, Z, and E introduced in Chapter 6.

Example 7.1. The operational semantics for the basic process algebra P is given by the fol-

lowing set of structural rules:

$$\frac{}{\mathsf{nil}\downarrow} \qquad \frac{}{!_a \xrightarrow{a} \mathsf{nil}} \qquad \frac{p \xrightarrow{a} p'}{p\,;q \xrightarrow{a} p'\,;q} \qquad \frac{p\downarrow \quad q \xrightarrow{a} q'}{p\,;q \xrightarrow{a} q'} \qquad \frac{p\downarrow \quad q\downarrow}{p\,;q\downarrow}$$

$$\frac{p \xrightarrow{a} p'}{p \sqcup q \xrightarrow{a} p'} \qquad \frac{q \xrightarrow{a} q'}{p \sqcup q \xrightarrow{a} q'} \qquad \frac{p\downarrow}{(p \sqcup q)\downarrow} \qquad \frac{q\downarrow}{(p \sqcup q)\downarrow}$$

The rules recursively define the relation $\rightarrow \subseteq P \times A \times P$ and a predicate $\downarrow \subseteq P$, on terms P and set of characters A. We write $p \xrightarrow{a} p'$ for $(p, a, p') \in \rightarrow$ and $p\downarrow$ for $p \in \downarrow$. Intuitively, the transition $p \xrightarrow{a} p'$ represents a term p which can evolve into term p' by printing the character a on the screen, whereas $p\downarrow$ holds for terms which can successfully terminate.

Example 7.2. The operational semantics for the language Z is given below. The rules recursively define a relation $\Downarrow \subseteq Z \times \mathbb{Z}$, where we write $t \Downarrow n$ for $(t, n) \in \Downarrow$. Intuitively, $t \Downarrow n$ means that term t can evaluate to integer n.

$$\frac{}{\mathsf{n} \Downarrow n} \qquad \frac{t \Downarrow n \quad u \Downarrow m}{t + u \Downarrow n + m} \qquad \frac{c \Downarrow 0 \quad t \Downarrow n}{\mathsf{ifz}\ c\ t\ e \Downarrow n} \qquad \frac{c \Downarrow n \quad n \neq 0 \quad e \Downarrow m}{\mathsf{ifz}\ c\ t\ e \Downarrow m}$$

Note that the semantics were given in a small-step style for P and a big-step style for Z. However, the mathematical approach to operational semantics of this chapter treats these two different styles uniformly, as long as the rules are of a particular type (see Section 7.3).

Example 7.3. The operational semantics for the language of exceptions E is given by the rules below and define a predicate $\uparrow \subseteq E$, where we write $e\uparrow$ for $e \in \uparrow$. Intuitively, $e\uparrow$ means that e can throw an exception.

$$\frac{}{\mathsf{throw}\uparrow} \qquad \frac{t\uparrow \quad u\uparrow}{(\mathsf{catch}\ t\ u)\uparrow}$$

As in Chapter 6, we will consider combining E with P and Z. In order to obtain an operational semantics for the combined language, not only do we need to put together the operational rules corresponding to each language, but also we need to add extra rules, for example, explaining how catch deals with the transitions defined by the other languages and how the operators in other languages deal with exceptions. In Figures 7.1, 7.2, and 7.3, we show all the rules that need to be added to combine E with P and Z.

There were some decisions to be made when combining these languages. For example, when combining P and Z (Figure 7.3), we decided that sequencing discards the value of its first argument, and that addition is evaluated from left to right (whereas before the order of evaluation did not matter).

More generally, combining operational semantics is not just a matter of the tedious and error-prone task of adding extra syntactical rules, but may also involve modifying the original rules, for example, to propagate state. This makes it difficult to formally relate the

$$\frac{p \xrightarrow{a} p'}{\mathsf{catch}\ p\ q \xrightarrow{a} \mathsf{catch}\ p'\ q} \qquad \frac{p\downarrow}{(\mathsf{catch}\ p\ q)\downarrow} \qquad \frac{p\uparrow \quad q \xrightarrow{a} q'}{\mathsf{catch}\ p\ q \xrightarrow{a} q'}$$

$$\frac{p\uparrow \quad q\downarrow}{(\mathsf{catch}\ p\ q)\downarrow} \qquad \frac{p\uparrow}{(p\,;q)\uparrow} \qquad \frac{p\downarrow \quad q\uparrow}{(p\,;q)\uparrow} \qquad \frac{p\uparrow}{(p \sqcup q)\uparrow} \qquad \frac{q\uparrow}{(p \sqcup q)\uparrow}$$

Figure 7.1: Additional rules for combining P and E

$$\frac{t \Downarrow n}{\mathsf{catch}\ t\ u \Downarrow n} \qquad \frac{t\uparrow \quad u \Downarrow n}{\mathsf{catch}\ t\ u \Downarrow n} \qquad \frac{t\uparrow}{t+u\uparrow} \qquad \frac{u\uparrow}{t+u\uparrow}$$

$$\frac{c\uparrow}{(\mathsf{ifz}\ c\ t\ e)\uparrow} \qquad \frac{c \Downarrow 0 \quad t\uparrow}{(\mathsf{ifz}\ c\ t\ e)\uparrow} \qquad \frac{c \Downarrow z \quad z \neq 0 \quad e\uparrow}{(\mathsf{ifz}\ c\ t\ e)\uparrow}$$

Figure 7.2: Additional rules for combining Z and E

$$\frac{p \Downarrow n \quad q \xrightarrow{a} q'}{(p\,;q) \xrightarrow{a} q'} \qquad \frac{p \Downarrow n \quad q\downarrow}{(p\,;q)\downarrow} \qquad \frac{p\downarrow \quad q \Downarrow n}{(p\,;q) \Downarrow n} \qquad \frac{p \Downarrow n}{(p \sqcup q) \Downarrow n} \qquad \frac{q \Downarrow n}{(p \sqcup q) \Downarrow n} \qquad \frac{p\downarrow}{p+q\downarrow}$$

$$\frac{p \Downarrow n \quad q\downarrow}{p+q\downarrow} \qquad \frac{p \xrightarrow{a} p'}{p+q \xrightarrow{a} p'+q} \qquad \frac{p \Downarrow n \quad q \xrightarrow{a} q'}{p+q \xrightarrow{a} p+q'} \qquad \frac{c \xrightarrow{a} c'}{(\mathsf{ifz}\ c\ t\ e) \xrightarrow{a} c'} \qquad \frac{c\downarrow}{(\mathsf{ifz}\ c\ t\ e)\downarrow}$$

$$\frac{c \Downarrow 0 \quad t\downarrow}{(\mathsf{ifz}\ c\ t\ e)\downarrow} \qquad \frac{c \Downarrow z \quad z \neq 0 \quad e\downarrow}{(\mathsf{ifz}\ c\ t\ e)\downarrow} \qquad \frac{c\downarrow}{(\mathsf{ifz}\ c\ t\ e)\downarrow} \qquad \frac{c \Downarrow 0 \quad t \xrightarrow{a} t'}{(\mathsf{ifz}\ c\ t\ e) \xrightarrow{a} t'} \qquad \frac{c \Downarrow z \quad z \neq 0 \quad e \xrightarrow{a} e'}{(\mathsf{ifz}\ c\ t\ e) \xrightarrow{a} e'}$$

Figure 7.3: Additional rules for combining Z and P

original and combined languages. The underlying problem is that SOS lacks a language-independent theory that would clarify what combining languages means in general, rather than for specific rules.

7.2 Transition Relations as Coalgebras

Operational semantics are given by a transition relation which represents execution steps in an abstract machine. Transition relations can be modeled in a generic, categorical way by coalgebras (Jacobs and Rutten 1997). Given an endofunctor B, a B-coalgebra is an object X and a structure map $X \to BX$. The carrier of the coalgebra X can be seen as the set of states of an abstract machine, while the endofunctor B represents the observable behaviour of the machine.

Every relation $R \subseteq X \times Y$ can be written as a function $X \to \mathcal{P}Y$ mapping every element

in X to its set of related elements in Y. The simplest technique for interpreting the powerset functor in Haskell is to use the list functor provided by the language. However, we will use the NonDet monad provided by the Monatron library instead. In this way we will benefit from the automatic lifting of its two algebraic operations zeroND and plusND (see Appendix A.4 for details of their implementation). Thus, we interpret relations $R \subseteq X \times Y$ as Haskell functions $\mathsf{X} \to \mathsf{NonDet}\ \mathsf{Y}$.

Example 7.4. The SOS rules for the language P define a transition relation $\rightarrow \subseteq P \times A \times P$ and a predicate $\downarrow \subseteq P$. The $\rightarrow$ relation is equivalent to a function of type $P \to \mathcal{P}(A \times P)$, and the predicate $\downarrow$ is equivalent to a function of type $P \to \mathsf{Bool} \cong \mathcal{P}(1)$. By the universal property of products, giving two such functions is equivalent to giving a function of type $P \to \mathcal{P}(A \times P) \times \mathcal{P}(1)$, which by the isomorphism $\mathcal{P}(A) \times \mathcal{P}(B) \cong \mathcal{P}(A + B)$, is in turn equivalent to a function

$$k : P \to \mathcal{P}(1 + A \times P).$$

That is, both transition relations can be given by a single coalgebra (P, k) for the functor $\mathcal{P}(1 + A \times -)$. In Haskell, we can express this functor as the following datatype, where Pr is the datatype $\mathsf{Pr}\ a = \mathsf{Pr}\ \mathsf{Char}\ a$ defined in Section 6.4.

```
data BP a = BP {unBP :: NonDet (Either () (Pr a))}

instance Functor BP where
  fmap f (BP m) = BP (fmap (fmap (fmap f)) m)
```

Example 7.5. Consider the language Z of Section 7.1. A simple inductive argument shows that the $\Downarrow$ relation is a function. Hence, we can describe the induced transition relation by a KI-coalgebra, where KI is the constant Int functor.

```
newtype KI a = KI Int
```

Example 7.6. The transition relation $\uparrow$ can be represented by a KE-coalgebra, where KE is the constant unit functor.

```
data KE a = KE
```

As shown in these last two examples, when the transition relation is a function, we can remove the powerset (or list) functor. In this manner, the determinism of the underlying transition system is made explicit, avoiding the need for a separate proof. Being able to describe precisely what is observable by choosing an appropriate behaviour functor is an important advantage of the coalgebraic approach.

Execution of transition systems

In order to execute a transition system specified by a coalgebra, we *unfold* the coalgebra (Hutton 1998; Jacobs and Rutten 1997) to construct a tree of observations. The appropriate notion of tree is given by the greatest fixpoint of the behaviour functor of the coalgebra.

$$
\begin{aligned}
&\textbf{codata}\ \mathsf{Nu}\ f = \mathsf{Nu}\ (f\ (\mathsf{Nu}\ f))\\
&\mathsf{out} \qquad :: (\mathsf{Nu}\ f) \to f\ (\mathsf{Nu}\ f)\\
&\mathsf{out}\ (\mathsf{Nu}\ \mathsf{n}) = \mathsf{n}\\
&\mathsf{unfold} \quad :: \mathsf{Functor}\ b \Rightarrow (x \to b\ x) \to x \to \mathsf{Nu}\ b\\
&\mathsf{unfold}\ g = \mathsf{Nu} \cdot \mathsf{fmap}\ (\mathsf{unfold}\ g) \cdot g
\end{aligned}
$$

As observed in Section 6.4, in Haskell there is no distinction between least and greatest fixpoint, and the use of **codata** above only expresses the intended meaning.

In conclusion, coalgebras provide an abstract model of transition systems, where the type of the transition system and its corresponding notion of equality are determined by a functor. However, as discussed in the next section, this is not sufficient to model structural operational semantics.

7.3 Mathematical Operational Semantics

Coalgebras provide an abstract model of transition systems. Unfortunately, they do not support a proper theory of SOS. In particular, the carrier of a coalgebra is unstructured, and hence a purely coalgebraic approach will not be able to take advantage of the fact that the carrier of the coalgebra is the set of terms, and hence, has an algebra structure. Therefore, in order to develop a mathematical operational semantics, what it is needed is a structure which contains both coalgebraic and algebraic features. Turi constructed such a structure in his categorical framework for SOS by focusing on the operational rules rather than on the transition relation.

In this section, we present our implementation of Turi's framework. To begin with, let us consider a typical operational rule and analyse its structure:

$$
\frac{p \xrightarrow{a} p'}{p\,;q \xrightarrow{a} p'\,;q} \qquad\qquad \frac{premisses}{source \to target}
$$

In general, a rule consists of some premisses and a conclusion. The source of the conclusion consists of an operator of the language (the ; operator, in the example above) applied to some metavariables (p and q) which stand for arbitrary terms. Premisses are transitions from these metavariables. Finally, the target of the conclusion is a term with metavariables taken from the source of the conclusion and from the premisses (q and p', respectively).

In the previous two sections, we showed how to abstract the notion of syntax by a signature functor and the notion of observable behaviour by a behaviour functor. Using these concepts we can abstract the structure of operational rules.

The Type of Operational Rules

Given a language with syntax determined by a signature functor s and behaviour functor b, its structural operational semantics is given by rules of the form:

```
type OR s b = ∀ x. s (x, b x) → b (Term s x)
```

The type above says that operational rules are defined by a polymorphic function which, given the source of the conclusion of a rule in which every variable is paired with its behaviour, it returns the transition in the conclusion of the rule.

Importantly, operational rules are polymorphic in the (meta)variables x in order to guarantee that the induced transition depends only on the behaviour of the subterms, and not on the actual subterms.

Example 7.7. We give an operational semantics to the constructs of P given in Section 6.1 with a behaviour functor BP.

```
orP                        :: OR P BP
orP Nil                    = BP (return (Left ()))
orP (Put c)                = BP (return (Right (Pr c nil)))
orP (Seq (_, bp) (q, bq))  = BP (unBP bp >>= either
                                 (λ() → unBP (fmap Var bq))
                                 (λ(Pr c p') → return (Right (Pr c (seq (Var p') (Var q))))))
orP (Alt (_, bp) (_, bq))  = BP (unBP (fmap Var bp) `plusND` unBP (fmap Var bq))
```

Function orP implements the operational rules of P given in Example 7.1 by pattern-matching on the operator in the source of the conclusion. In the case of Nil, the only possible transition is to terminate. In the case of Put c, the only possible transition is to print c and then behave as the term nil. In the case of Seq p q, the type of each possible transition of p is analysed in order to see which transition to perform. If p may terminate, then Seq p q may continue execution with the behaviour of q. If p may print c and continue execution with term p', then Seq p q may print c and continue execution with term Seq p' q. In the case of Alt p q, the possible transitions are the union of the possible transitions from p and the possible transitions from q.

Operational rules OR not only are a structured, language-independent formulation of SOS, but also have the important property that they are guaranteed to induce a transition

relation with bisimulation as a congruence and to generate an adequate denotational model, as shown in the next subsection.

Obtaining a Transition Relation

Every operational rule $\mathsf{OR}\ s\ b$ induces a lifting opMonad of the syntax monad $\mathsf{Term}\ s$ to the category of b-coalgebras. The function opMonad (the operational monad of Turi (1996)) takes a b-coalgebra on x and returns a b-coalgebra on $\mathsf{Term}\ s\ x$. Intuitively, opMonad shows that given an operational rule and the semantics of variables x in the terms, we can give semantics to terms with variables from x.

$$\begin{aligned}
&\mathsf{opMonad} \quad :: (\mathsf{Functor}\ s, \mathsf{Functor}\ b) \Rightarrow \\
&\qquad\qquad \mathsf{OR}\ s\ b \to (x \to b\ x) \to \mathsf{Term}\ s\ x \to b\ (\mathsf{Term}\ s\ x) \\
&\mathsf{opMonad}\ op\ k = \mathsf{snd} \cdot \mathsf{foldTerm}\ \langle \mathsf{Var}, \mathsf{fmap}\ \mathsf{Var} \cdot k \rangle \\
&\qquad\qquad\qquad\qquad\qquad \langle \mathsf{Con} \cdot \mathsf{fmap}\ \mathsf{fst}, \mathsf{fmap}\ \mathsf{join} \cdot op \rangle \\
&\qquad\qquad \textbf{where}\ \langle f, g \rangle\ a = (f\ a, g\ a)
\end{aligned}$$

In order to execute a Program (where $\mathsf{Program}\ s = \mathsf{Term}\ s\ \mathsf{Empty}$ as defined in the previous chapter) we unfold the coalgebra obtained from opMonad:

$$\begin{aligned}
&\mathsf{run} \quad :: (\mathsf{Functor}\ s, \mathsf{Functor}\ b) \Rightarrow \mathsf{OR}\ s\ b \to \mathsf{Program}\ s \to \mathsf{Nu}\ b \\
&\mathsf{run}\ op = \mathsf{unfold}\ (\mathsf{opMonad}\ op\ \mathsf{empty})
\end{aligned}$$

Moreover, an operational rule gives rise to a denotational model $s\ (\mathsf{Nu}\ b) \to (\mathsf{Nu}\ b)$.

$$\begin{aligned}
&\mathsf{denModel} \quad :: (\mathsf{Functor}\ s, \mathsf{Functor}\ b) \Rightarrow \mathsf{OR}\ s\ b \to s\ (\mathsf{Nu}\ b) \to \mathsf{Nu}\ b \\
&\mathsf{denModel}\ or = \mathsf{unfold}\ (\mathsf{opMonad}\ or\ \mathsf{out}) \cdot \mathsf{Con} \cdot \mathsf{fmap}\ \mathsf{Var} \\
&\mathsf{eval} \quad :: (\mathsf{Functor}\ s, \mathsf{Functor}\ b) \Rightarrow \mathsf{OR}\ s\ b \to \mathsf{Program}\ s \to \mathsf{Nu}\ b \\
&\mathsf{eval}\ or \quad = \mathsf{foldTerm}\ \mathsf{empty}\ (\mathsf{denModel}\ or)
\end{aligned}$$

Theorem 7.8 (Adequacy (Turi 1996))**.** *The operational and the denotational semantics induced by an operational rule or coincide.*

$$\mathsf{run}\ or \quad \equiv \quad \mathsf{eval}\ or$$

Corollary 7.9. *Bisimulation is a congruence for the transition relation corresponding to an operational rule or.*

This concludes our functional implementation of Turi's mathematical operational semantics. In the next section, we tackle the question of how to modularly combine operational rules.

7.4 Modular Operational Semantics

Operational rules $\mathsf{OR}\ s\ b$ are defined for a given signature functor s and behaviour functor b. In Section 6.1 we showed how to obtain modular syntax by abstracting from a specific

signature functor. Our goal now is to abstract from specific behaviour functors in order to obtain modular behaviours. We achieve this by structuring the behaviour functor with three components:

- A monad m which models computational effects;
- A step functor f which determines the notion of a (small) step;
- Final values v.

$$\textbf{type}\ \mathsf{B}\ m\ f\ v\ x = m\ (\mathsf{Either}\ v\ (f\ x))$$

Additionally, we use the $\overset{*}{\hookrightarrow}$ relation to structure values in v (when several types of values are present) and the $\hookrightarrow$ relation to structure the step functor f (when several types of steps are present).

Putting together modular behaviours with modular syntax yields the following definition of *modular operational rules*:

$$\textbf{type}\ \mathsf{MOR}\ s\ t\ m\ f\ v = \forall\ x.\ s\ (x, \mathsf{B}\ m\ f\ v\ x) \to \mathsf{B}\ m\ f\ v\ (\mathsf{Term}\ t\ x)$$

Modular operational rules MOR differ from concrete operational rules OR in two ways:

(1) there is a distinction between the signature s of the language being defined and the signature of the complete language t, simplifying the combination of modular operational rules (otherwise, one needs to traverse terms, inserting each construct into the total language);

(2) behaviours are structured with a monad m, a step functor f and a type of values v.

Proposition 7.10. *The carrier of the final coalgebra on a structured behaviour functor* $(\mathsf{B}\ m\ f\ v)$ *coincides with the monad transformer of* f*-steps applied to the monad* m *at* v*.*

$$\mathsf{StepT}\ f\ m\ v \qquad \cong \qquad \mathsf{Nu}\ (\mathsf{B}\ m\ f\ v)$$

Proof. Unfold the definitions. □

Given a MOR, we can ossify[1] it and obtain a concrete OR by fixing the signature of the complete language to be the signature of the language being defined, and by providing a behaviour which satisfies the behaviour requirements of the given MOR. A new datatype BF is introduced for the technical reason that B was defined as a type synonym, and hence cannot be made an instance of the class Functor (as needed by opMonad, for example).

$$\textbf{newtype}\ \mathsf{BF}\ m\ f\ v\ y = \mathsf{BF}\ \{\mathsf{unBF} :: m\ (\mathsf{Either}\ v\ (f\ y))\}$$

[1] To *ossify* is to turn into bone, and figuratively, to become rigid.

```
instance (Monad m, Functor f) ⇒ Functor (BF m f v) where
  fmap f (BF m) = BF (liftM (fmap (fmap f)) m)
ossify      :: (Functor s) ⇒ MOR s s m f v → OR s (BF m f v)
ossify mor = BF · mor · fmap (λ(a, b) → (a, unBF b))
```

Defining Modular Operational Rules

We give examples of modular operational rules. For convenience, we first define several auxiliary functions.

```
caseAny      :: (Monad m, Functor f) ⇒
                B m f v x → (Term s x → Term s x) →
                (v → B m f v (Term s x)) → B m f v (Term s x)
caseAny b t h = b ≫= either h (return · Right · fmap (t · Var))

caseVal      :: (Monad m, Functor f, u ↪* v) ⇒
                B m f v x → (Term s x → Term s x) →
                (u → B m f v (Term s x)) → B m f v (Term s x)
caseVal b t h = caseAny b t (λv → case (prj* v) of
                                    Nothing → return (Left v)
                                    Just u  → h u)
```

The function caseAny takes a structured behaviour and does a case analysis trying to find any value v. If a value is found, the third argument h –which handles values– is applied. On the other hand, if a step is found, the step is performed and computation is continued by applying the endofunction on Terms t. The function caseVal is similar but only handles a particular type of value $u \overset{*}{\hookrightarrow} v$. (cf. the operation $\gg\!=^{\hookrightarrow}$ in 6.1).

Finally, we define functions val and stp, which return a structured behaviour from a value or a step respectively, and the function up, which takes a structured behaviour on a variable and returns a structured behaviour on a term.

```
val :: (u ↪* v, Monad m) ⇒ u → B m f v x
val = return · Left · inj*

stp :: (g ↪ f, Monad m) ⇒ g x → B m f v x
stp = return · Right · inj

up :: (Monad m, Functor s, Functor f) ⇒ B m f v x → B m f v (Term s x)
up = liftM (fmap (fmap Var))
```

Using these tools we can carry out one of the fundamental ideas of the approach: a modular language should have the least possible requirements on syntax and behaviour, as illustrated in the following example.

Example 7.11. The semantics of P as a modular operational rule is:

$$
\begin{aligned}
&\mathsf{morP} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ s, \mathsf{P} \hookrightarrow s, () \overset{*}{\hookrightarrow} v, \mathsf{Pr} \hookrightarrow f, \mathsf{MonadAlgModel}\ \mathsf{NonDetOp}\ m) \\
&\qquad\qquad \Rightarrow \mathsf{MOR}\ \mathsf{P}\ s\ m\ f\ v \\
&\mathsf{morP}\ \mathsf{Nil} \qquad\qquad = \mathsf{val}\ () \\
&\mathsf{morP}\ (\mathsf{Put}\ c) \qquad\qquad = \mathsf{stp}\ (\mathsf{Pr}\ c\ \mathsf{nil}) \\
&\mathsf{morP}\ (\mathsf{Seq}\ (\_, bp)\ (q, bq)) = \mathsf{caseAny}\ bp\ (\text{`}seq\text{`}\ (\mathsf{Var}\ q))\ (\lambda\_ \rightarrow \mathsf{up}\ bq) \\
&\mathsf{morP}\ (\mathsf{Alt}\ (\_, bp)\ (\_, bq)) = \mathsf{up}\ (bp\ \text{`}\mathsf{plusND}\text{`}\ bq)
\end{aligned}
$$

The operational rule morP requires:

- $\mathsf{P} \hookrightarrow s$: The signature functor s should include the constructs of P;
- $() \overset{*}{\hookrightarrow} v$: The type of values should include the unit type $()$;
- $\mathsf{Pr} \hookrightarrow f$: The step functor f should include the functor Pr;
- MonadAlgModel NonDetOp m : The monad m should support the algebraic operations zeroND and plusND for non-determinism.

In the sequencing construct, the computation moves to second argument after *any* value is reached. Alternatively, one could have used caseVal and only move to the second argument when a value $()$ is reached. This would mean that if p reaches a value other than $()$ then that value would be returned without ever executing q. Remarkably, Seq does not need to know anything about the step functor; computation simply continues until a value is reached.

Example 7.12. The modular operational semantics for the language Z is given by morZ, where the monad in the structured behaviour forces the choice of an order of evaluation of the arguments of Add.

$$
\begin{aligned}
&\mathsf{morZ} :: (\mathsf{Functor}\ f, \mathsf{Functor}\ s, \mathsf{Monad}\ m, \mathsf{Z} \hookrightarrow s, \mathsf{Int} \overset{*}{\hookrightarrow} v) \Rightarrow \mathsf{MOR}\ \mathsf{Z}\ s\ m\ f\ v \\
&\mathsf{morZ}\ (\mathsf{Num}\ i) \qquad\qquad = \mathsf{val}\ i \\
&\mathsf{morZ}\ (\mathsf{Add}\ (p, bp)\ (q, bq)) \qquad = \mathsf{caseVal}\ bp\ (\text{`}add\text{`}\ \mathsf{Var}\ q) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda i \rightarrow \mathsf{caseVal}\ bq\ (\mathsf{Var}\ p\ \text{`}add\text{`}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\lambda j \rightarrow \mathsf{val}\ (i + j :: \mathsf{Int}))) \\
&\mathsf{morZ}\ (\mathsf{Ifz}\ (\_, bc)\ (t, bt)\ (e, be)) = \mathsf{caseVal}\ bc\ (\text{`}\mathsf{ifz}\text{`}\ (\mathsf{Var}\ t, \mathsf{Var}\ e)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda i \rightarrow \textbf{if}\ i \equiv (0 :: \mathsf{Int})\ \textbf{then}\ \mathsf{up}\ bt\ \textbf{else}\ \mathsf{up}\ be)
\end{aligned}
$$

The operational rule morZ requires:

- $\mathsf{Z} \hookrightarrow s$: The signature functor s should include the constructs of Z;
- $\mathsf{Int} \overset{*}{\hookrightarrow} v$: The type of values should include the type of integers Int.

Example 7.13. The modular operational semantics for the language E is given by morE.

```
morE :: (Functor s, Functor f, E ↪ s, MonadAlgModel (ThrowOp ()) m,
         MonadModel (HandleOp ()) m) ⇒ MOR E s m f v
morE Thr                 = throw ()
morE (Cat (_, bp) (q, bq)) = handle (bp >>= return · fmap (fmap g)) (λ() → up bq)
                               where g p' = Var p' 'cat' Var q
```

As opposed to the interpreter for E in Section 6.3, the semantics of Cat is not just a matter of using the operation handle provided by the monad m. It is also necessary to catch the exceptions that might be thrown in future steps.

The operational rule morE requires:

- E ↪ s : The signature functor s should include the constructs of E;
- MonadAlgModel (ThrowOp ()) m : The monad m should support the algebraic operation throw for throwing exceptions of type ();
- MonadModel (HandleOp ()) m : The monad m should support the operation handle for handling exceptions of type ().

Example 7.14. As a last example we define an operational rule for the merge operator. Since behaviours already provide a notion of step, the only requirement on the behaviour is that the monad m should support non-determinism.

```
morR :: (Functor s, Functor f, R ↪ s, MonadAlgModel NonDetOp m) ⇒ MOR R s m f v
morR (Par (p, bp) (q, bq)) = caseAny bp (| Var q) (λ_ → up bq)
                               'plusND'
                               liftM (fmap (fmap (λy → Var p | Var y))) bq
```

7.5 Combining Modular Operational Rules

Combining modular operational rules is a simple matter of taking their copair.

```
(⩊)        :: MOR s t m f v → MOR s' t m f v → MOR (s ⊕ s') t m f v
op1 ⩊ op2 = copair op1 op2
```

This is the fundamental tool for combining modular operational rules. The constraint that the monad m and behaviour b should be the same for the input rules of ⩊ appears to be a severe restriction that undermines our original goal. However, since MOR are expected to be written on an abstract notion of structured behaviour with certain requirements, the requirements on the behaviour of the combined rules is the combination of the requirements on behaviour of each of the operational rules being combined.

Example 7.15. We construct an operational rule corresponding to the combination of the modular operational rules morP, morZ, and morE. The requirements on syntax and behaviour

of $(\mathsf{morP} \Cup \mathsf{morZ} \Cup \mathsf{morE})$ are the combination of the requirements of each language. To obtain a concrete operational semantics for L we fix the syntax to be $\mathsf{L} = \mathsf{P} \oplus \mathsf{Z} \oplus \mathsf{E}$ and we instantiate the behaviour to the monad $\mathsf{ExcT}\ ()\ \mathsf{NonDet}$, step functor Pr, and values to be either a successfully terminating process $()$ or an integer.

The operational rule morR does not add any requirements, except for R being present in the signature functor.

$$\mathsf{orL} :: \mathsf{OR}\ (\mathsf{P} \oplus \mathsf{Z} \oplus \mathsf{E})\ (\mathsf{BF}\ (\mathsf{ExcT}\ ()\ \mathsf{NonDet})\ \mathsf{Pr}\ (\mathsf{Either}\ ()\ \mathsf{Int}))$$
$$\mathsf{orL} = \mathsf{ossify}\ (\mathsf{morP} \Cup \mathsf{morZ} \Cup \mathsf{morE})$$
$$\mathsf{orR} :: \mathsf{OR}\ (\mathsf{R} \oplus \mathsf{P} \oplus \mathsf{Z} \oplus \mathsf{E})\ (\mathsf{BF}\ (\mathsf{ExcT}\ ()\ \mathsf{NonDet})\ \mathsf{Pr}\ (\mathsf{Either}\ ()\ \mathsf{Int}))$$
$$\mathsf{orR} = \mathsf{ossify}\ (\mathsf{morR} \Cup \mathsf{morP} \Cup \mathsf{morZ} \Cup \mathsf{morE})$$

Example 7.16. Adding exceptions to P is just a matter of adding the semantics of throw and catch. In this case, we do not need values to contain integers.

$$\mathsf{ep} :: \mathsf{OR}\ (\mathsf{E} \oplus \mathsf{P})\ (\mathsf{BF}\ (\mathsf{ExcT}\ ()\ \mathsf{NonDet})\ \mathsf{Pr}\ ())$$
$$\mathsf{ep} = \mathsf{ossify}\ (\mathsf{morE} \Cup \mathsf{morP})$$

Example 7.17. Adding exceptions to Z is once again, just a matter of adding the semantics of throw and catch. In this case, we choose the monad to be the exception monad and, since there are no requirements on the step functor, we choose it to be the identity functor.

$$\mathsf{ez} :: \mathsf{OR}\ (\mathsf{E} \oplus \mathsf{Z})\ (\mathsf{BF}\ (\mathsf{Exception}\ ())\ \mathsf{Id}\ \mathsf{Int})$$
$$\mathsf{ez} = \mathsf{ossify}\ (\mathsf{morE} \Cup \mathsf{morZ})$$

In order to obtain a combined semantics we need to provide a monad which supports the operations required by the modular components. Since the requirements do not specify any order on the layering of effects, there could be many different monads that satisfy these requirements, each yielding different combined semantics, as it was the case with the modular interpreters of Chapter 6.

An advantage of defining the combination operation for MOR rather than OR is that elements of MOR are flexible enough to allow the separate definition of operators which depend on other operators. This flexibility is especially advantageous if each operator has different requirements on the behaviour functor, as each operator will be defined with less requirements, yielding a more general semantics. For example, with MORs it is possible to define the construct nil and put from P separately, while with ORs this is not possible since put depends on nil.

7.6 Summary

We have developed a modular approach to operational semantics which allows us to define the semantics of a language as a combination of the semantics of its individual components.

The approach is based on writing the operational semantics on partially known syntax and behaviour, and on the representation of an operational semantics as a polymorphic function that distributes syntax over behaviour. This high-level modular approach leads to a simple and natural implementation in Haskell.

The modular operational semantics obtained in this chapter and the modular interpreters of the previous chapter have some interesting differences.

- The modular operational semantics have a concept of "step" which is more natural than in the case of modular interpreters, where the StepT transformer had to be introduced in order to make possible the definition of merge.
- The interaction with "global" exceptions is different. In the case of modular interpreters, it was obtained by applying the exception monad transformer last, while for modular operational semantics the exception monad transformer was applied to each step. The difference here is noticeable when handling exceptions. In the interpreters case, the handle operation supplied by the monad is enough, while in the case of operational rules, extra work is required (but it is still possible to define it).
- In the interpreter of parallel merge, the operation caseStep for doing case analysis on StepT had to be lifted manually. In the operational rule, this was not necessary.

A practical approach to modular operational semantics for certain specific effects has recently been put forward by Mosses (2004) (see also Mosses and New 2008), but it is based on the syntactic rather than semantic approach to SOS.

Turi (1997) showed with a few examples how operational rules which are parametric in their behaviour could be instantiated to different settings but did not attempt to systematize this technique. Lenisa, Power, and Watanabe (2000) defined an operation that combines two operational rules on the same behaviour $\mathsf{OR}\ s\ b$ and $\mathsf{OR}\ s'\ b$ into an operational rule $\mathsf{OR}\ (s \oplus s')\ b$, but did not consider the problem of semantics with different behaviour.

Kick and Power (2004) presented the dual of the syntax combination operation for ORs, that is, an operation which takes two operational rules $\mathsf{OR}\ s\ b$ and $\mathsf{OR}\ s\ b'$, and returns a $\mathsf{OR}\ s\ (b \otimes b')$ (where $\otimes$ is the functorial product). This operation does not seem to be powerful enough to support the combinations we are trying to obtain. However, it would be interesting to see how this operation, in the particular case of the behaviour being of the form $\mathcal{P}B$, could be used to obtain results similar to ours by exploiting the isomorphism $\mathcal{P}(A + B) \cong \mathcal{P}(A) \times \mathcal{P}(B)$.

Some languages require the framework to be interpreted in *CPO*-like categories (Klin 2004), in particular for dealing with general recursion, but for all the examples in this chapter, the structure of *Set* would be enough.

Part III

Conclusion

Chapter 8

Summary and Further Work

In this chapter we review the contributions of this thesis and discuss some directions for future research.

8.1 Summary of Contributions

This thesis generalizes and extends the incremental approach to modular monadic semantics. The generalization is obtained by working with monoids in a monoidal category, rather than monads, and considering operations associated to a monoid $\hat{M}$ to be maps $H\hat{M} \to M$ for a functor $H : \mathsf{Mon}(\hat{\mathcal{E}}) \to \mathcal{E}$. Working at this level of generality, the incremental approach is then extended with general results about lifting of operations. These results show how an operation can be lifted through a monoid transformer according to the classes the operation and the transformer belong to. The lifting results are summarised in the following table[1], along with the relevant classes of operations and transformers.

Lifting Theorem	Algebraic Theorem 3.5	Codensity Theorem 4.15	Monoidal Theorem 4.9
Transformer	Any	Functorial	Monoidal
Operation	$S \longrightarrow M$	$S \otimes M \longrightarrow M$	$(S \otimes M) \otimes F \longrightarrow M$

Many of the usual effect-manipulating operations that are usually associated with computational monads—in fact, all operations from (Liang, Hudak, and Jones 1995)—are shown to be definable from operations which fall into one of these classes. In particular, and perhaps most surprisingly, the callcc operation is definable in terms of algebraic operations. This shows that it is well-behaved and that it can be lifted along any monad morphism.

[1]The Algebraic Theorem is given for algebraic operations and the Codensity Theorem requires a monoidal right-closed category.

The theory is implemented in Monatron, a monad transformer library for Haskell. The library solves some problems in existing monad transformer libraries by incorporating uniform liftings, and by making the models of the operations first-class entities (as opposed to type-class instances). The usage of the library is demonstrated with an implementation of modular interpreters.

Turi's semantics-based approach to operational semantics induces well-behaved transition systems (where bisimulation is a congruence) and an internally fully-abstract denotational model. We have implemented Turi's semantics for the first time, allowing programmers to write small-step operational semantics in a natural manner, while keeping the benefits of denotational semantics. This approach to operational semantics was extended with a technique for writing operational semantics modularly, which structures the behaviour functor with a monad, a step functor, and an object of values. One can then apply the incremental approach to modular monadic semantics to structure the monad in the behaviour functor.

8.2 Directions for Further Research

The notion of operation presented in this thesis leaves out some operations such as elim and case (introduced in Chapter 3.2). A problem in operations such as elim, which capture initiality, is that the transformed monad might not carry an initial algebra. A possible solution might arise from the following observations.

As shown by Uustalu (2003), the free monad and the free completely iterative monad over a functor Σ arise as the least/greatest fixpoint of a functor $F : \mathcal{C} \to \mathbf{Mon}(\mathcal{C})$. For example, let the underlying monoidal category $\mathcal{C}$ be a category of endofunctors and let F be the functor $F(Y)\,X \mathrel{\hat{=}} X + \Sigma Y$. Then $\mu Y.\, F(Y)\,X$ is the free monad over Σ on X, i.e. $\mu Y.\, X + \Sigma Y$ (assuming the fixpoint exists).

Given any functor $F : \mathcal{C} \to \mathbf{Mon}(\mathcal{C})$, and a covariant monad transformer $\hat{T} = (T, \mathsf{lift}^T)$, one can obtain a *transformed functor*:

$$F^T \mathrel{\hat{=}} T \circ F : \mathcal{C} \to \mathbf{Mon}(\mathcal{C})$$

Then, $M^T X \mathrel{\hat{=}} \mu Y.F^T(Y)X$ can be considered as the monad $MX = \mu Y.F(Y)X$ with the addition of T-effects. In general, the resulting monad M^T will be different from the monad obtained by applying $\hat{T}$ and then the step monad transformer, or the monad obtained by applying first the step monad transformer and then $\hat{T}$.

By transforming monads arising from functors $F : \mathcal{C} \to \mathbf{Mon}(\mathcal{C})$ in this way, the transformed monad still carries an initial algebra, and the operations associated to $\hat{T}$ can still be applied, so it seems like a promising alternative, but details still need to be worked out.

This approach might also shed light on how to lift handlers of operations (Plotkin and Pretnar 2009), since these are defined by initiality of the free model of an algebraic theory.

Another application might be to provide a structured foundation for semantics of inheritance in object-oriented systems (Cook and Palsberg 1994).

There seems to be a relation between the algebraic operations for a signature S presented in this thesis and the functorial terms of arity S for equational systems (Fiore and Hur 2009). A topic of future work is to make this connection precise and use free constructions for equational systems to define monad transformers by adding new operations satisfying certain equations to a pre-existing monad.

Another application of the connection with equational systems would be to define an equational system for continuations using the algebraic operations callcc and throw and an appropriate set of equations.

Apart from extending the approach to more expressive operations (as discussed above) one might also consider extending the techniques to other transformers. For example, by providing a lifting result for mixed-variant monad transformers such as the codensity and continuation monad transformer.

Monads are not the only structure proposed to model computational effects. Also other structures such as arrows (Hughes 2000), the related Freyd categories (Jacobs and Hasuo 2006; Power and Robinson 1997), and comonads (Uustalu and Vene 2005) have been proposed for this purpose. Arrows can be viewed as monoids in suitable monoidal categories (Heunen and Jacobs 2006) so our lifting results are directly applicable but the details need to be worked out. Comonads are the dual of monads, so one could try to dualise the theory to comonoids, and apply it to the case of comonads. The extension to arrows and comonads could then be implemented in Haskell and incorporated to the library Monatron as arrow transformers and comonad transformers.

Our implementation of operational semantics, as Turi's original work, is fundamentally first-order. Therefore, it would be interesting to consider modular operational semantics for languages with more advanced features such as binding. Incorporating binding operations into Turi's framework is a difficult task, see (Fiore and Staton 2004; Fiore and Turi 2001) but it should be possible.

Additionally, it would be interesting to implement a modular operational semantics framework in a theorem prover, so as to formally reason about the properties of the combined semantics with the help of a machine.

The definition of modular operational rules is done in the Haskell language. It would be interesting to define a more restricted language, so that operational rules can be defined in a syntax closer to the original (syntactical) rules (perhaps taking some ideas from Mosses (2004)). Additionally, one would like to be able to print the combination of two operational semantics as operational rules. With a more restricted language, this should be easier than it is now, where reifying a Haskell function is required.

For some languages, such as PCF with algebraic effects (Plotkin and Power 2001*a*), it

seems that the intended semantics require us to interpret the operational rules in the Kleisli category of a monad, where operational rules are natural transformations:

$$\Sigma(\mathsf{Id} \times B) \xrightarrow{\bullet} MBT_\Sigma$$

Consequently, one can ask under which conditions it is possible to develop Turi's semantics in such a Kleisli category. For example, we would need to calculate a final coalgebra in the Kleisli category of a monad. Sufficient conditions for its existence are given by Hasuo, Jacobs, and Sokolova (2007).

8.3 Conclusion

This thesis generalises and extends the incremental approach to modular monadic semantics with uniform liftings of operations. It starts with the development of theoretical foundations in category-theory, and then applies the abstract results to obtain a concrete, usable implementation of a monad transformer library in Haskell.

The well-known ability of category theory to provide higher-level concepts that abstract from irrelevant details was crucial in developing the theory. The expressivity of Haskell was essential to carrying out the implementation. However, there are many subtleties and pitfalls that make the passage of ideas from one to the other a hazardous journey. Computer science would clearly benefit enormously from a practical programming language overtly rooted in category theory.

Appendix A

Monatron Source Code

A.1 Monad Transformers

```
class MonadT t where
  treturn :: Monad m ⇒ a → t m a
  treturn = lift · return
  tbind   :: Monad m ⇒ t m a → (a → t m b) → t m b
  lift    :: Monad m ⇒ m a → t m a

class MonadT t ⇒ FMonadT t where
  tmap :: (Functor m, Functor n) ⇒ (∀ b. m b → n b) → t m a → t n a

class FMonadT t ⇒ MMonadT t where
  flift       :: Functor f ⇒ f a → t f a   -- should coincide with lift!
  monoidalT :: (Functor f, Functor g) ⇒ t f (t g a) → t (f ∘ g) a
```

Codensity Monad Transformer

```
newtype Cod f a = Cod {unCod :: ∀ b. (a → f b) → f b}

down  :: Monad m ⇒ Cod m a → m a
down c = unCod c return

instance MonadT Cod where
  tbind c f = Cod (λk → unCod c (λa → unCod (f a) k))
  lift m    = Cod (m>>=)

instance Monad m ⇒ Monad (Cod m) where
  return = treturn
  (>>=) = tbind

instance Functor f ⇒ Functor (Cod f) where
  fmap f (Cod c) = Cod $ λk → c (k · f)
```

State Monad Transformer

```
newtype StateT s m a = S {unS :: s → m (a, s)}

runStateT     :: s → StateT s m a → m (a, s)
runStateT s m = unS m s

instance MonadT (StateT s) where
  tbind m k   = S (λs → unS m s >>= λ(a, s') → unS (k a) s')
  lift m      = S (λs → m >>= λa → return (a, s))

instance FMonadT (StateT s) where
  tmap f (S m) = S (f · m)

instance MMonadT (StateT s) where
  flift t              = S (λs → fmap (λa → (a, s)) t)
  monoidalT (S t) = S (λs → Comp $ fmap (λ(S t', s') → t' s') (t s))

instance Monad m ⇒ Monad (StateT s m) where
  return = treturn
  (>>=) = tbind

instance Functor m ⇒ Functor (StateT s m) where
  fmap f (S g) = S (λs → fmap (λ(a, s') → (f a, s')) (g s))
```

Reader Monad Transformer

```
newtype ReaderT s m a = R {unR :: s → m a}

runReaderT     :: s → ReaderT s m a → m a
runReaderT s m = unR m s

instance MonadT (ReaderT s) where
  tbind m k = R (λs → unR m s >>= λa → unR (k a) s)
  lift m    = R (λ_ → m)

instance FMonadT (ReaderT s) where
  tmap f (R m) = R (f · m)

instance Monad m ⇒ Monad (ReaderT s m) where
  return = treturn
  (>>=) = tbind

instance MMonadT (ReaderT s) where
  flift t              = R (λ_ → t)
  monoidalT (R t) = R (λs → Comp $ fmap (($s) · unR) (t s))

instance Functor m ⇒ Functor (ReaderT s m) where
  fmap f (R g) = R (λs → fmap f (g s))
```

Exception Monad Transformer

```
newtype ExcT x m a = X { unX :: m (Either x a) }

runExcT :: ExcT x m a → m (Either x a)
runExcT = unX

instance MonadT (ExcT x) where
  tbind (X m) f = X (do a ← m
                        case a of Left x → return (Left x)
                                  Right b → unX (f b))
  lift m        = X (liftM Right m)

instance FMonadT (ExcT x) where
  tmap f = X · f · unX

instance Monad m ⇒ Monad (ExcT x m) where
  return  = treturn
  (>>=)   = tbind

instance Functor m ⇒ Functor (ExcT x m) where
  fmap f (X m) = X (fmap (either Left (Right.f)) m)
```

Writer Monad Transformer

```
newtype WriterT w m a = W { unW :: m (a, w) }

runWriterT :: WriterT w m a → m (a, w)
runWriterT = unW

instance Monoid w ⇒ MonadT (WriterT w) where
  tbind (W m) f = W (do (a, w) ← m
                        (a', w') ← unW (f a)
                        return (a', w `mappend` w'))
  lift m        = W (liftM (λa → (a, mempty)) m)

instance Monoid w ⇒ FMonadT (WriterT w) where
  tmap f    = W · f · unW

instance Monoid w ⇒ MMonadT (WriterT w) where
  flift t               = W (fmap (λa → (a, mempty)) t)
  monoidalT (W t) = W $ Comp $ fmap (λ(W t', w) →
                                fmap (λ(a, w') → (a, w `mappend` w')) t') $ t

instance (Monad m, Monoid w) ⇒ Monad (WriterT w m) where
  return    = treturn
  (>>=)     = tbind

instance Functor m ⇒ Functor (WriterT x m) where
```

```
fmap f (W m) = W (fmap (λ(a,w) → (f a,w)) m)
```

Continuation Monad Transformer

```
newtype ContT r m a = C {unC :: (a → m r) → m r}
runContT    :: Monad m ⇒ (a → m r) → ContT r m a → m r
runContT k m = unC m k
instance MonadT (ContT r) where
  tbind c f = C (λk → unC c (λa → unC (f a) k))
  lift m    = C (m>>=)
instance Monad m ⇒ Monad (ContT r m) where
  return = treturn
  (>>=) = tbind
```

Step Monad Transformer

```
newtype StepT f m x = T {runT :: m (Either x (f (StepT f m x)))}
instance (Functor f, Monad m) ⇒ Monad (StepT f m) where
  return = treturn
  (>>=) = tbind
instance (Functor f, Monad m) ⇒ Functor (StepT f m) where fmap = liftM
caseStep          :: (Functor f, Monad m) ⇒
                     (a → StepT f m x) → (f (StepT f m a) → StepT f m x)
                      → StepT f m a → StepT f m x
caseStep v c (T m) = T (m >>= either (runT · v) (runT · c))
unfoldStepT    :: (Functor f, Monad m) ⇒ (y → m (Either x (f y))) → y → StepT f m x
unfoldStepT k y = T (liftM (fmap (fmap (unfoldStepT k))) (k y))
instance (Functor f) ⇒ MonadT (StepT f) where
  tbind t f = caseStep f (T · return · Right · fmap (‘tbind‘f)) t
  lift      = T · liftM Left
instance (Functor f) ⇒ FMonadT (StepT f) where
  tmap t (T m) = T (t (fmap (either Left (Right · fmap (tmap t))) m))
```

Non-Determinism Monad Transformer

```
data NDSig f a = NilT | ConsT a (f a)
instance Functor f ⇒ Functor (NDSig f) where
  fmap _ NilT        = NilT
```

```
  fmap f (ConsT a fa) = ConsT (f a) (fmap f fa)
newtype NonDetT m a = L { unL :: m (NDSig (NonDetT m) a) }
runNonDetT :: NonDetT m a → m (NDSig (NonDetT m) a)
runNonDetT = unL
emptyL :: Monad m ⇒ NonDetT m a
emptyL = L $ return $ NilT
appendL :: Monad m ⇒ NonDetT m a → NonDetT m a → NonDetT m a
appendL (L m1) (L m2) = L $ do l  ← m1
                               case l of
                                 NilT       → m2
                                 ConsT a l1 → return (ConsT a (appendL l1 (L m2)))
foldNonDetT :: Monad m ⇒ (a → m b → m b) → m b → NonDetT m a → m b
foldNonDetT c n (L m) = do l ← m
                           case l of
                             NilT       → n
                             ConsT a l1 → c a (foldNonDetT c n l1)
collectNonDetT  :: Monad m ⇒ NonDetT m a → m [a]
collectNonDetT lt = foldNonDetT (λa m → m >>= return · (a:)) (return []) lt
instance MonadT NonDetT where
  lift m      = L $ liftM (`ConsT`emptyL) m
  m `tbind` f = L $ foldNonDetT (λa l → unL $ f a `appendL` L l)
                                (return NilT)
                                m
instance FMonadT NonDetT where
  tmap t (L m) = L $ t $ fmap (λlsig → case lsig of
                                 NilT       → NilT
                                 ConsT a l → ConsT a (tmap t l)) m
instance Monad m ⇒ Monad (NonDetT m) where
  return      = treturn
  (>>=)       = tbind
instance Functor f ⇒ Functor (NonDetT f) where
  fmap h (L f) = L $ fmap (fmap h) f
```

A.2 Monads

```
type State s      = StateT s Id
```

```
runState s          = runId · runStateT s
type Reader s       = ReaderT s Id
runReader e         = runId · runReaderT e
type Writer w       = WriterT w Id
runWriter           = runId · runWriterT
type Exception x    = ExcT x Id
runException        = runId · runExcT
type Cont r         = ContT r Id
runCont k           = runId. runContT (Id. k)
type NonDet         = NonDetT Id
runNonDet           = runId. runNonDetT
```

Identity Monad

```
newtype Id a = Id a
runId          :: Id a → a
runId (Id a)    = a
instance Monad Id where
  return        = Id
  (Id a) >>= f  = f a
instance Functor Id where fmap = liftM
```

A.3 Models and Standard Liftings

```
type ExtModel f g m = ∀ a. f (m (g a)) → m a
type Model f m      = ∀ a. f (m a) → m a
type AlgModel f m   = ∀ a. f a → m a
toAlg      :: (Functor f, Monad m) ⇒ Model f m → AlgModel f (Cod m)
toAlg op t  = Cod $ λk → op (fmap k t)
liftModel  :: (Functor f, Monad m, Functor m, FMonadT t, Monad (t (Cod m))) ⇒
              Model f m → Model f (t m)
liftModel op = tmap down · join · lift · toAlg op · fmap (tmap lift)
liftAlgModel  :: (MonadT t, Monad m, Functor f) ⇒ AlgModel f m → AlgModel f (t m)
liftAlgModel op = lift · op
liftExtModel  ::  (Functor f, Functor g, Monad m, Functor m,
                   MMonadT t, Functor (t f), Functor (t m)) ⇒
```

```
                                ExtModel f g m → ExtModel f g (t m)
liftExtModel op = tmap (op · fmap deComp · deComp) ·
                  monoidalT · flift · fmap (monoidalT · fmap flift)
```

A.4 Operations

State Operations

```
data StateOp s a = Get (s → a) | Put s a

instance Functor (StateOp s) where
  fmap f (Get g)   = Get (f · g)
  fmap f (Put s a) = Put s (f a)

modelStateT              :: Monad m ⇒ AlgModel (StateOp s) (StateT s m)
modelStateT (Get g)   = S (λs → return (g s,s))
modelStateT (Put s a) = S (λ_ → return (a,s))

getX    :: Monad m ⇒ AlgModel (StateOp s) m → m s
getX op = op $ Get id

putX     :: Monad m ⇒ AlgModel (StateOp s) m → s → m ()
putX op s = op $ Put s ()
```

Reader Operations

```
data ReaderOp s a = Ask (s → a) | InEnv s a

instance Functor (ReaderOp s) where
  fmap f (Ask g)     = Ask (f · g)
  fmap f (InEnv s a) = InEnv s (f a)

modelReaderT              :: Monad m ⇒ Model (ReaderOp s) (ReaderT s m)
modelReaderT (Ask g)      = R (λs → runReaderT s (g s))
modelReaderT (InEnv s a) = R (λ_ → runReaderT s a)

askX    :: Monad m ⇒ Model (ReaderOp s) m → m s
askX op = op $ Ask return

inEnvX          :: Monad m ⇒ Model (ReaderOp s) m → s → m a → m a
inEnvX op s m = op $ InEnv s m
```

Exception Operations

```
data ThrowOp x a  = Throw x
data HandleOp x a = Handle a (x → a)

instance Functor (ThrowOp x) where
```

```
  fmap _ (Throw x) = Throw x
instance Functor (HandleOp x) where
  fmap f (Handle a h) = Handle (f a) (f · h)
modelThrowExcT              :: Monad m ⇒ AlgModel (ThrowOp x) (ExcT x m)
modelThrowExcT (Throw x) = X (return (Left x))
modelHandleExcT             :: Monad m ⇒ Model (HandleOp x) (ExcT x m)
modelHandleExcT (Handle m h) = X (unX m ≫= λexa → case exa of
                                    Left x → unX (h x)
                                    Right a → return (Right a))
throwX      :: Monad m ⇒ AlgModel (ThrowOp x) m → x → m a
throwX op x = op $ Throw x
handleX     :: Monad m ⇒ Model (HandleOp x) m → m a → (x → m a) → m a
handleX op m h = op $ Handle m h
```

Writer Operations

```
data WriterOp w a = Trace w a
instance Functor (WriterOp w) where
  fmap f (Trace w a) = Trace w (f a)
modelWriterT :: (Monad m, Monoid w) ⇒ AlgModel (WriterOp w) (WriterT w m)
modelWriterT (Trace w a) = W (return (a, w))
traceX      :: (Monad m) ⇒ AlgModel (WriterOp w) m → w → m ()
traceX op w = op $ Trace w ()
```

Continuation Operations

```
data ContOp r a = Abort r | CallCC ((a → r) → a)
instance Functor (ContOp r) where
  fmap _ (Abort r)  = Abort r
  fmap f (CallCC k) = CallCC (λc → f (k (c · f)))
modelContT              :: Monad m ⇒ AlgModel (ContOp (m r)) (ContT r m)
modelContT (Abort mr) = C $ λ_ → mr
modelContT (CallCC k) = C $ λc → c (k c)
abortX      :: Monad m ⇒ AlgModel (ContOp r) m → r → m a
abortX op r = op (Abort r)
callCCX     :: Monad m ⇒ AlgModel (ContOp r) m → ((a → r) → a) → m a
callCCX op f = op (CallCC f)
```

```
callccX      :: Monad m ⇒ AlgModel (ContOp r) m → ((a → m b) → m a) → m a
callccX op f  = join $ callCCX op (λk → f (λx → abortX op (k (return x))))
```

Step Operations

```
newtype StepOp f x = StepOp (f x)

instance (Functor f) ⇒ Functor (StepOp f) where
  fmap h (StepOp fa) = StepOp (fmap h fa)

modelStepT                :: (Functor f, Monad m) ⇒ Model (StepOp f) (StepT f m)
modelStepT (StepOp fa) = T (return (Right fa))

stepX    :: (Monad m) ⇒ Model (StepOp f) m → f (m x) → m x
stepX op = op · StepOp
```

Non-Determinism Operations

```
data NonDetOp a = ZeroND | PlusND a a

instance Functor NonDetOp where
  fmap _ ZeroND       = ZeroND
  fmap f (PlusND a b) = PlusND (f a) (f b)

modelNonDetT                :: Monad m ⇒ AlgModel NonDetOp (NonDetT m)
modelNonDetT ZeroND         = emptyL
modelNonDetT (PlusND t u) = appendL (return t) (return u)

zeroNDX      :: Monad m ⇒ AlgModel NonDetOp m → m a
zeroNDX op   = op ZeroND

plusNDX      :: Monad m ⇒ AlgModel NonDetOp m → m a → m a → m a
plusNDX op t u = join $ op (PlusND t u)
```

A.5 Automatic Liftings

Classes of Models

```
class (Functor f, Monad m) ⇒ MonadModel f m where
  model :: Model f m

class (Functor f, Monad m) ⇒ MonadAlgModel f m where
  algModel :: AlgModel f m

class (Functor f, Functor g, Monad m) ⇒ MonadExtModel f g m where
  extModel :: ExtModel f g m
```

Automatic Lifting of Models

```
instance (FMonadT t, MonadModel f m, Monad (t m), Functor m, Monad (t (Cod m)))
         ⇒ MonadModel f (t m) where
  model = liftModel model

instance (MonadT t, MonadAlgModel f m, Monad (t m))
         ⇒ MonadAlgModel f (t m) where
  algModel = liftAlgModel algModel

instance (MMonadT t, MonadExtModel f g m, Monad (t m), Functor m,
          Functor (t f), Functor (t m)) ⇒ MonadExtModel f g (t m) where
  extModel = liftExtModel extModel
```

Reader

```
ask :: MonadModel (ReaderOp s) m ⇒ m s
ask = askX model

inEnv :: MonadModel (ReaderOp s) m ⇒ s → m a → m a
inEnv = inEnvX model

instance Monad m ⇒ MonadModel (ReaderOp s) (ReaderT s m) where
  model = modelReaderT
```

State

```
get :: MonadAlgModel (StateOp s) m ⇒ m s
get = getX algModel

put :: MonadAlgModel (StateOp s) m ⇒ s → m ()
put = putX algModel

instance Monad m ⇒ MonadAlgModel (StateOp s) (StateT s m) where
  algModel = modelStateT
```

Continuations

```
abort :: MonadAlgModel (ContOp r) m ⇒ r → m a
abort = abortX algModel

callCC :: MonadAlgModel (ContOp r) m ⇒ ((a → r) → a) → m a
callCC = callCCX algModel

instance Monad m ⇒ MonadAlgModel (ContOp (m r)) (ContT r m) where
  algModel = modelContT
```

Exceptions

```
throw :: MonadAlgModel (ThrowOp x) m ⇒ x → m a
throw = throwX algModel

handle :: MonadModel (HandleOp x) m ⇒ m a → (x → m a) → m a
handle = handleX model

instance Monad m ⇒ MonadModel (HandleOp x) (ExcT x m) where
  model = modelHandleExcT

instance Monad m ⇒ MonadAlgModel (ThrowOp x) (ExcT x m) where
  algModel = modelThrowExcT
```

Writer

```
trace :: MonadAlgModel (WriterOp w) m ⇒ w → m ()
trace = traceX algModel

instance (Monoid w, Monad m) ⇒ MonadAlgModel (WriterOp w) (WriterT w m) where
  algModel = modelWriterT
```

Step

```
step :: (Functor f, MonadModel (StepOp f) m) ⇒ f (m x) → m x
step = stepX model

instance (Functor f, Monad m) ⇒ MonadModel (StepOp f) (StepT f m) where
  model = modelStepT
```

Nondeterminism

```
zeroND :: MonadAlgModel NonDetOp m ⇒ m a
zeroND = zeroNDX algModel

plusND :: MonadAlgModel NonDetOp m ⇒ m a → m a → m a
plusND = plusNDX algModel

instance Monad m ⇒ MonadAlgModel NonDetOp (NonDetT m) where
  algModel = modelNonDetT
```

A.6 Other

Functor Composition

```
newtype (f ∘ g) a = Comp { deComp :: (f (g a)) }
```

```
instance (Functor f, Functor g) ⇒ Functor (f ∘ g) where
  fmap f (Comp fga) = Comp (fmap (fmap f) fga)
```

Either **Functor Instance**

```
instance Functor (Either x) where
  fmap _ (Left x)  = Left x
  fmap f (Right y) = Right (f y)
```

Bibliography

Abbott, Michael, Thorsten Altenkirch, and Neil Ghani (2003). "Categories of Containers". In: *FoSSaCS*. Ed. by Andrew D. Gordon. Vol. 2620. Lecture Notes in Computer Science. Springer, pp. 23–38.

Aceto, Luca, Wan Fokkink, and Chris Verhoef (2001). "Structural Operational Semantics". In: *Handbook of Process Algebra*. Ed. by Jan A. Bergstra, Alban Ponse, and Scott A. Smolka. New York, USA: Elsevier Science Inc., pp. 197–292.

Adámek, Jiří and Jiří Rosický (1994). *Locally Presentable and Accessible Categories*. London Mathematical Society Lecture Notes. Cambridge University Press.

Asperti, Andrea and Giuseppe Longo (1990). *Categories, Types and Structures*. MIT Press.

Bainbridge, Edwin S. et al. (1990). "Functorial Polymorphism". In: *Theoretical Computer Science* 70.1, pp. 35–64.

Barr, Michael and Charles Wells (1985). *Toposes, Triples and Theories*. Vol. 278. Grundlehren der Mathematischen Wissenschaften. New York: Springer-Verlag.

— (1995). *Category Theory for Computer Science*. Prentice Hall.

Benton, Nick, John Hughes, and Eugenio Moggi (2000). "Monads and Effects". In: *International Summer School On Applied Semantics APPSEM2000*. Springer-Verlag, pp. 42–122.

Bergstra, Jan A., Wan Fokkink, and Alban Ponse (2001). "Process Algebra with Recursive Operations". In: *Handbook of Process Algebra*. Ed. by Jan A. Bergstra, Alban Ponse, and Scott A. Smolka. New York, USA: Elsevier Science Inc., pp. 333–389.

Bergstra, Jan A. and Jan Willem Klop (1985). "Algebra of Communicating Processes with Abstraction". In: *Theoretical Computer Science* 37, pp. 77–121.

Bergstra, Jan A., Alban Ponse, and Scott A. Smolka, eds. (2001). *Handbook of Process Algebra*. New York, USA: Elsevier Science Inc.

Bird, Richard (1998). *Introduction to Functional Programming*. 2nd ed. Prentice Hall.

Borceux, Francis (1994*a*). *Handbook of Categorical Algebra. Basic Category Theory*. Vol. 1. Cambridge University Press.

— (1994*b*). *Handbook of Categorical Algebra. Categories and Structures*. Vol. 2. Cambridge University Press.

Castagna, Giuseppe, ed. (2009). *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009*. Vol. 5502. Lecture Notes in Computer Science. Springer.

Claessen, Koen (2004). "Parallel Parsing Processes". In: *Journal of Functional Programming* 14.6, pp. 741–757.

Cook, William R. and Jens Palsberg (1994). "A Denotational Semantics of Inheritance and Its Correctness". In: *Information and Computation* 114.2 (Nov. 1994), pp. 329–350.

Diatchki, Iavor. *MonadLib*. URL: `http://www.galois.com/~diatchki/monadLib/` (visited on 06/06/2009).

Espinosa, David (1993). "Semantic Lego". Unpublished manuscript. Dec. 1993.

— (1994). "Building Interpreters by Transforming Stratified Monads".

— (1995). "Semantic Lego". PhD thesis. Columbia University.

Filinski, Andrzej (1994). "Representing Monads". In: *POPL*, pp. 446–457.

Fiore, Marcelo and Chung-Kil Hur (2009). "On the construction of free algebras for equational systems". In: *Theoretical Computer Science* 410.18, pp. 1704–1729.

Fiore, Marcelo and Sam Staton (2004). "Comparing Operational Models of Name-Passing Process Calculi." In: *Electronic Notes in Theoretical Computer Science* 106, pp. 91–104.

Fiore, Marcelo and Daniele Turi (2001). "Semantics of Name and Value Passing". In: *Proc. 16th LICS Conf.* IEEE. Computer Society Press, pp. 93–104.

Fokkink, Wan (2000). *Introduction to Process Algebra*. Secaucus, NJ, USA: Springer-Verlag New York, Inc.

Ghani, Neil and Christoph Lüth (1997). "Monads and Modular Term Rewriting". In: *Proceedings of CTCS'97*. Lecture Notes in Computer Science 1290. Springer-Verlag, pp. 69–86.

Ghezzi, Carlos, Mehdi Jazayeri, and Dino Mandrioli (2002). *Fundamentals of Software Engineering*. 2nd ed. Prentice Hall.

Gibbons, Jeremy, ed. (2009). *IFIP Working Conference on Domain Specific Languages*. Springer, LNCS.

Gill, Andy. *Monad Transformer Library*. URL: `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mtl-1.1.0.2` (visited on 06/06/2009).

Girard, Jean-Yves (1972). "Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur". PhD thesis. Université Paris 7.

Glasgow Haskell Compiler. URL: `http://haskell.org/ghc/` (visited on 06/06/2009).

Hagino, Tatsuya (1987). "A Categorical Programming Language". PhD thesis. University of Edinburgh.

Haskell Platform (2009). URL: `http://hackage.haskell.org/platform/` (visited on 06/06/2009).

Hasuo, Ichiro, Bart Jacobs, and Ana Sokolova (2007). "Generic Trace Semantics via Coinduction". In: *Logical Methods in Computer Science* 3.4.

Heunen, Chris and Bart Jacobs (2006). "Arrows, like Monads, are Monoids". In: *Electronic Notes in Theoretical Computer Science* 158, pp. 219–236.

Hinze, Ralf (2000). "Deriving Backtracking Monad Transformers". In: *ICFP*, pp. 186–197.

Hughes, John (1986). "A Novel Representation of Lists and its Application to the Function "reverse"". In: *Information Processing Letters* 22.3, pp. 141–144.

— (1995). "The Design of a Pretty-printing Library". In: *Advanced Functional Programming*. Ed. by J. Jeuring and E. Meijer. Springer Verlag, LNCS 925, pp. 53–96.

— (2000). "Generalising Monads to Arrows". In: *Science of Computer Programming* 37.1-3 (May 2000), pp. 67–111.

Hutton, Graham (1998). "Fold and Unfold for Program Semantics". In: *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming*. Baltimore, Maryland.

— (2007). *Programming in Haskell*. Cambridge University Press.

Hutton, Graham and Erik Meijer (1998). "Monadic Parsing in Haskell". In: *Journal of Functional Programming* 8.4 (July 1998), pp. 437–444.

Hyland, J. Martin E. (1988). "A Small Complete Category". In: *Journal of Pure and Applied Logic* 40, pp. 135–165.

Hyland, Martin, Gordon D. Plotkin, and John Power (2006). "Combining Effects: Sum and Tensor". In: *Theoretical Computer Science* 357.1-3, pp. 70–99.

Hyland, Martin and John Power (2007). "The Category Theoretic Understanding of Universal Algebra: Lawvere Theories and Monads". In: *Electronic Notes in Theoretical Computer Science* 172, pp. 437–458.

Jacobs, Bart and Ichiro Hasuo (2006). "Freyd is Kleisli, for arrows". In: *Proceedings of the Workshop on Mathematically Structured Functional Programming*.

Jacobs, Bart and Jan Rutten (1997). "A Tutorial on (Co)Algebras and (Co)Induction". In: *Bulletin of the European Association for Theoretical Computer Science* 62, pp. 222–259.

Jaskelioff, Mauro. *Modular Monad Transformer Library*. URL: `http://hackage.haskell.org/cgi-bin/hackage-scripts/package/mmtl` (visited on 06/06/2009).

— (2008). "Monatron: an Extensible Monad Transformer Library". In: *Implementation and Application of Functional Languages*. Accepted for publication.

— (2009). "Modular Monad Transformers". In: *European Symposium on Programming*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, pp. 64–79.

Jaskelioff, Mauro, Neil Ghani, and Graham Hutton (2008). "Modularity and Implementation of Mathematical Operational Semantics". In: *Proceedings of the Workshop on Mathematically Structured Functional Programming*. Reykjavik, Iceland.

Jaskelioff, Mauro and Eugenio Moggi (2009). "Monad Transformers as Monoid Transformers". In: *Theoretical Computer Science*. Submitted for publication.

Johann, Patricia and Janis Voigtländer (2004). "Free Theorems in the Presence of *seq*". In: *POPL*, pp. 99–110.

— (2009). "A Family of Syntactic Logical Relations for the Semantics of Haskell-like languages". In: *Information and Computation* 207.2, pp. 341–368.

Jones, Mark P. and Luc Duponcheel (1993). *Composing Monads*. Tech. rep. YALEU/DCS/RR-1004. New Haven, Connecticut: Yale University.

Kelly, G. Maxwell (1980). "A Unified Treatment of Transfinite Constructions for Free Algebras, Free Monoids, Colimits, Associated Sheaves, and so on". In: *Bulletin of the Australian Mathematical Society* 22.01, pp. 1–83.

Kelly, G. Maxwell and John Power (1993). "Adjunctions Whose Counits are Coequalizers and Presentations of Finitary Monads". In: *Journal of Pure and Applied Algebra* 89.1–2, pp. 163–179.

Kick, Marco and A. John Power (2004). "Modularity of Behaviours for Mathematical Operational Semantics." In: *Electronic Notes in Theoretical Computer Science* 106, pp. 185–200.

King, David J. and Philip Wadler (1992). "Combining Monads". In: *Functional Programming*. Ed. by John Launchbury and Patrick M. Sansom. Workshops in Computing. Springer, pp. 134–143. ISBN: 3-540-19820-2.

Klin, Bartek (2004). "Adding Recursive Constructs to Bialgebraic Semantics". In: *Journal of Logic and Algebraic Programming* 60-61.

Lawvere, F. William (1963). "Funtorial Semantics of Algebraic Theories and Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theories". Reprints in Theory and Applications of Categories, 5 (2004) 1-121. PhD thesis. Columbia University.

Lenisa, Marina, John Power, and Hiroshi Watanabe (2000). "Distributivity for Endofunctors, Pointed and Co-Pointed Endofunctors, Monads and Comonads". In: *Proceedings 3rd Workshop on Coalgebraic Methods in Computer Science, CMCS'00, Berlin, Germany, 25–26 March 2000*. Ed. by Horst Reichel. Vol. 33. Amsterdam: Elsevier.

Liang, Sheng and Paul Hudak (1996). "Modular Denotational Semantics for Compiler Construction". In: *ESOP*. Ed. by Hanne Riis Nielson. Vol. 1058. Lecture Notes in Computer Science. Springer, pp. 219–234.

Liang, Sheng, Paul Hudak, and Mark Jones (1995). "Monad Transformers and Modular Interpreters". In: *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: San Francisco, California, January 22–25, 1995*. Ed. by ACM. New York, NY, USA: ACM Press, pp. 333–343.

Lüth, Christoph and Neil Ghani (2002). "Composing Monads Using Coproducts". In: *International Conference on Functional Programming*. Vol. 37. 9, pp. 133–144.

Mac Lane, Saunders (1971). *Categories for the Working Mathematician*. Graduate Texts in Mathematics 5. Second edition, 1998. Springer-Verlag.

Manes, Ernie G. (1976). *Algebraic Theories*. Springer-Verlag.

— (1998). "Implementing Collection Classes with Monads". In: *Mathematical Structures in Computer Science* 8.3, pp. 231–276. ISSN: 0960-1295.

Meijer, Erik, Maarten Fokkinga, and Ross Paterson (1991). "Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire". In: *Proceedings of the 5th ACM Confer-*

ence on Functional Programming Languages and Computer Architecture. New York, NY, USA: Springer-Verlag New York, Inc., pp. 124–144.

Milius, Stefan (2005). "Completely Iterative Algebras and Completely Iterative Monads". In: *Information and Computation* 196.1, pp. 1 –41.

Moggi, Eugenio (1989*a*). *An Abstract View of Programming Languages*. Tech. rep. ECS-LFCS-90-113. Edinburgh, Scotland: Edinburgh University.

— (1989*b*). "Computational Lambda-Calculus and Monads". In: *LICS*. IEEE Computer Society, pp. 14–23.

— (1991). "Notions of Computation and Monads". In: *Information and Computation* 93.1, pp. 55–92.

— (1997). "Metalanguages and Applications". In: *Semantics and Logics of Computation*. Publications of the Newton Institute. CUP.

Mosses, Peter D. (2004). "Modular Structural Operational Semantics". In: *Journal of Logic and Algebraic Programming* 60–61. Special issue on SOS, pp. 195–228.

Mosses, Peter D. and Mark J. New (2008). "Implicit Propagation in Structural Operational Semantics". In: *SOS 2008, Preliminary Proceedings*. Final version to appear in ENTCS.

Papaspyrou, Nikolaos S. (2001). "A Resumption Monad Transformer and its Applications in the Semantics of Concurrency". In: *Proceedings of the 3rd Panhellenic Logic Symposium*. Anogia, Greece.

Peyton Jones, Simon L., ed. (2003). *Haskell 98 Language and Libraries: the Revised Report*. Cambridge University Press.

Peyton Jones, Simon L. and Philip Wadler (1993). "Imperative Functional Programming". In: *POPL*, pp. 71–84.

Peyton Jones, Simon, Andrew Gordon, and Sigbjorn Finne (1996). "Concurrent Haskell". In: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. New York, NY, USA: ACM, pp. 295–308.

Peyton Jones, Simon et al. (2007). "Practical type inference for arbitrary-rank types". In: *Journal of Functional Programming* 17.1, pp. 1–82.

Plotkin, Gordon D. (1981). *A Structural Approach to Operational Semantics*. Tech. rep. DAIMI FN-19. University of Aarhus.

— (2004). "A Structural Approach to Operational Semantics". In: *Journal of Logic and Algebraic Programming* 60-61, pp. 17–139.

Plotkin, Gordon D. and John Power (2001*a*). "Adequacy for Algebraic Effects". In: *Lecture Notes in Computer Science* 2030, pp. 1+.

— (2001*b*). "Semantics for Algebraic Operations". In: *Electronic Notes in Theoretical Computer Science* 45.

Plotkin, Gordon D. and John Power (2002). "Notions of Computation Determine Monads". In: *FoSSaCS*. Ed. by Mogens Nielsen and Uffe Engberg. Vol. 2303. Lecture Notes in Computer Science. Springer, pp. 342–356.

— (2004). "Computational Effects and Operations: An Overview". In: *Electronic Notes in Theoretical Computer Science* 73, pp. 149–163.

Plotkin, Gordon D. and Matija Pretnar (2009). "Handlers of Algebraic Effects". In: *ESOP*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, pp. 80–94.

Power, John and Edmund Robinson (1997). "Premonoidal Categories and Notions of Computation". In: *Mathematical Structures in Computer Science* 7.5, pp. 453–468.

Reynolds, John C. (1974). "Towards a theory of type structure". In: *Proc. Colloque sur la Programmation*. Vol. 19. Springer, pp. 408–425.

— (1980). "Using Category Theory to Design Implicit Conversions and Generic Operators". In: *Semantics-Directed Compiler Generation*. Ed. by Neil D. Jones. Vol. 94. Lecture Notes in Computer Science. Springer, pp. 211–258.

— (1998). *Theories of Programming Languages*. Cambridge University Press.

Reynolds, John C. and Gordon D. Plotkin (1993). "On Functors Expressible in the Polymorphic Typed Lambda Calculus". In: *Information and Computation* 105.1, pp. 1–29.

Rezk, Charles (1996). "Spaces of Algebra Structures and Cohomology of Operads". PhD thesis. Massachusetts Institute of Technology.

Stewart, Don and Spencer Sjanssen (2007). "Xmonad". In: *Proceedings of the ACM SIGPLAN workshop on Haskell*. ACM, pp. 119–119.

Swierstra, Wouter (2008). "Data Types à la Carte". In: *Journal of Functional Programming* 18.3, pp. 1–14.

Thompson, Simon (1999). *Haskell: The Craft of Functional Programming*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Turi, Daniele (1996). "Functorial Operational Semantics and its Denotational Dual". PhD thesis. Free University, Amsterdam.

— (1997). "Categorical Modelling of Structural Operational Rules: Case Studies". In: *Category Theory and Computer Science*, pp. 127–146.

Turi, Daniele and Gordon D. Plotkin (1997). "Towards a Mathematical Operational Semantics". In: *Proceedings of the 12th LICS Conference*. IEEE. Computer Society Press, pp. 280–291.

Uustalu, Tarmo (2003). "Generalizing Substitution". In: *Theoretical Informatics and Applications* 37.4, pp. 315–336.

Uustalu, Tarmo and Varmo Vene (2005). "Signals and Comonads". In: *Journal of Universal Computer Science* 11.7, pp. 1311–1326.

Voigtländer, Janis (2008). “Asymptotic Improvement of Computations over Free Monads”. In: *MPC*, pp. 388–403.

Wadler, Philip (1989). “Theorems for Free!” In: *Functional Programming Languages and Computer Architecture*. ACM Press, pp. 347–359.

— (1992). “The Essence of Functional Programming”. In: *POPL*, pp. 1–14.