A Coalgebraic Semantics for Imperative Programming Languages

Faris Abou-Saleh

Submitted in part fulfilment of the requirements for the degree of Doctor of Philosophy in Computing of the University of London and the Diploma of Imperial College, 2nd October, 2013
I declare that this thesis is my own work, and that all else is explicitly referenced.

Faris Abou-Saleh

March 1, 2014

The copyright of this thesis rests with the author and is made available under a Creative Commons Attribution Non-Commercial No Derivatives licence. Researchers are free to copy, distribute or transmit the thesis on the condition that they attribute it, that they do not use it for commercial purposes and that they do not alter, transform or build upon it. For any reuse or redistribution, researchers must make clear to others the licence terms of this work.
Abstract

In the theory of programming languages, one often takes two complementary perspectives. In operational semantics, one defines and reasons about the behaviour of programs; and in denotational semantics, one abstracts away implementation details, and reasons about programs as mathematical objects or denotations. The denotational semantics should be compositional, meaning that denotations of programs are determined by the denotations of their parts. It should also be adequate with respect to operational equivalence: programs with the same denotation should be behaviourally indistinguishable.

One often has to prove adequacy and compositionality independently for different languages, and the proofs are often laborious and repetitive. These proofs were provided systematically in the context of process algebras by the mathematical operational semantics framework of Turi and Plotkin – which represented transition systems as coalgebras, and program syntax by free algebras; operational specifications were given by distributive laws of syntax over behaviour. By framing the semantics on this abstract level, one derives denotational and operational semantics which are guaranteed to be adequate and compositional for a wide variety of examples.

However, despite speculation on the possibility, it is hard to apply the framework to programming languages, because one obtains undesirably fine-grained behavioural equivalences, and unconventional notions of operational semantics. Moreover, the behaviour of these languages is often formalised in a different way – such as computational effects, which may be thought of as an interface between programs and external factors such as non-determinism or a variable store; and comodels, or transition systems which implement these effects.

This thesis adapts the mathematical operational semantics framework to provide semantics for various classes of programming languages. After identifying the need for such an adaptation, we show how program behaviour may be characterised by final coalgebras in suitably order-enriched Kleisli categories. We define both operational and denotational semantics, first for languages with syntactic effects, and then for languages with effects and/or comodels given by a Lawvere theory. To ensure adequacy and compositionality, we define concrete and abstract operational rule-formats for these languages, based on the idea of evaluation-in-context; we give syntactic and then categorical proofs that those properties are guaranteed by operational specifications in these rule-formats.
Acknowledgements

I would like to thank my supervisor, Dirk Pattinson, for his guidance and support throughout the years of my PhD. Over the course of many discussions, and during the many setbacks I encountered on the way, he was always patient, thoughtful, and willing to go out of his way to help with whatever requests and doubts I had. His insights and advice helped to motivate me and to take a step back whenever I was stuck, and offered encouragement when it seemed like all options had been exhausted.

I would also like to thank my second supervisor, Philippa Gardner, for her kind attention and willingness to provide assistance, or simply a sympathetic ear, when it was needed. She gave me valuable feedback and wise advice about my academic concerns outside the project, and helped me to take a wider perspective on how my research was developing.

I owe a large debt to the many academics who kindly shared their knowledge and experience with me, whether by email, or in our conversations in workshops, conferences, pubs, and restaurants. I could not possibly name them all, but some of the academics I would like to thank are Andrej Bauer, Ichiro Hasuo, Martin Hyland, Paul Levy, Jean-Paul Melliès, Gordon Plotkin, John Power, Alex Simpson, and Sam Staton, for their generosity.

I am also grateful for the friendship of the post-doctoral researchers I have met, and my fellow PhD students, for sharing their thoughts and giving me a chance to share my research woes; and also for reminding me that the wider world still existed. In particular I would like to thank Rob Myers for the many conversations we had, and his frequent encouragement. I also thank Nick Bezhanishvilli, Ingo Battenfeld, Tzu-Chun Chen, Liang-Ting Chen, Fredrik Dahlqvist, Victor Faion, Tzu-Ching Horng, Raymond Hu, Clemens Kupke, Björn Lellman, Reuben Rowe, and Florian Widmann for their companionship in the last four years.

Finally, I would like to thank my family for their patience and constant support. They were always there for me in countless moments when I needed them, during what was a long and sometimes arduous journey. It would have been much harder without their help.
Contents

Abstract 3

Acknowledgements 5

1 Introduction 13

1.1 Background ................................................................. 13

1.1.1 Transition Systems Are Coalgebras ................................ 13

1.1.2 Process Algebra and Programming Language Semantics ...... 14

1.1.3 Notions of Computation and Effects ............................... 16

1.2 Goals of this Thesis ....................................................... 17

1.3 Thesis Contributions and Structure .................................... 18

1.4 Related Work ............................................................... 20

2 Background and Preliminaries 22

2.1 Mathematical Operational Semantics ................................. 22

2.1.1 Syntax and Algebras .................................................. 22

2.1.2 The Free Syntax Functor $T$ ....................................... 29
2.1.3 Free Constructions, Adjunctions, and Monads . . . . . . . . . . . . . . . 32
2.1.4 Monadic Strength and Enrichment . . . . . . . . . . . . . . . . . . . . . 33
2.1.5 Distributivity . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
2.1.6 Behaviour and Coalgebras . . . . . . . . . . . . . . . . . . . . . . . . . . 35
2.1.7 Combining Operational and Denotational Semantics via the Final Coal-
gebra . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
2.1.8 Operational Models From Abstract Operational Semantics (aOS) . . . 40
2.1.9 Adequacy and Compositionality . . . . . . . . . . . . . . . . . . . . . . . . 44
2.2 Trace Semantics via Kleisli Coalgebras . . . . . . . . . . . . . . . . . . . . . 45
  2.2.1 Kleisli Categories . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46
  2.2.2 Coalgebras in a Kleisli Category . . . . . . . . . . . . . . . . . . . . . . . 47
  2.2.3 Constructing the Final Kleisli Coalgebra . . . . . . . . . . . . . . . . . . 50
2.3 Effects, State, and Comodels . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
  2.3.1 Monads and Notions of Computation . . . . . . . . . . . . . . . . . . . . . 58
  2.3.2 Computational Effects . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60
  2.3.3 State via Comodels . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65
  2.3.4 Constraints on Comodels . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

3 MOS in a Mixed Kleisli Category for Syntactic Effects 69
  3.1 Applying MOS To Programming Languages Directly . . . . . . . . . . . . . . . 69
    3.1.1 Functorial Behaviour . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
    3.1.2 Operational Rules and Models . . . . . . . . . . . . . . . . . . . . . . . 75
3.1.3 Behavioural Equivalence and the Final Kleisli Coalgebra

3.2 Introducing a Mixed Kleisli Setting

3.2.1 Effect Syntax and Behaviour

3.3 A Semantic Domain for Syntactic Effects, through Order-Enrichment

3.3.1 The Final $T_e B$-coalgebra in the Context of Syntactic Effects

3.3.2 The Semantic Domain in a Kleisli category

3.3.3 The Base Category of $\text{Cpo}_{\perp_!}$

3.3.4 From Effectless to Effectful Operational Specifications

3.4 Adequacy and Compositionality in a Mixed Kleisli Setting

3.4.1 Fine- and Coarse-grained Semantic Domains for Effects

3.4.2 Adapting Adequacy and Compositionality from Fine- to Coarse-grained Semantics

3.4.3 Reducing Adequacy and Compositionality to a Condition of Cones

3.4.4 A Syntactic Proof of the Condition of Cones, Under the eEIC Format

4 Semantics for Comodels and Effects

4.1 Effectful and Comodel-based Languages As Coalgebras

4.1.1 The Semantic Effectful Transition System (ets)

4.1.2 Converting a Syntactic ETS into a Semantic ETS

4.1.3 The Comodel-Based Transition System (cts)

4.1.4 Converting an ETS Into a CTS

4.1.5 The Comodel-and-Effect Based Transition System (cets)
4.1.6 Converting an $\text{ETS}$ into a semantic $\text{CETS}$ ........................................ 146

4.2 Evaluation-In-Context Congruence (EIC) formats for Effectful and Stateful Languages ........................................................................................................... 149

4.2.1 Evaluation-in-Context for $\text{CTS}$, $\text{CETS}$, and $\text{ETS}$: EIC 1-3 ............... 151

4.2.2 Unifying the Formats: The Abstract EIC Format ........................................... 160

4.3 Semantic Domains and Behavioural Equivalence ............................................. 164

4.3.1 Semantics and Behavioural Equivalence for $\text{CTS}$’s .................................. 164

4.3.2 Syntactic and Semantic $\text{ETS}$’s ................................................................. 165

4.3.3 Syntactic and Semantic $\text{CETS}$’s ............................................................... 168

4.4 Adequacy and Compositionality for Stateful and Effectful Languages .......... 170

4.4.1 Denotational Models .................................................................................... 170

4.4.2 A Monadic Approach to Adequacy and Compositionality ....................... 172

4.4.3 Proof of Theorem 4.4.4 ............................................................................... 178

5 Discussion ............................................................................................................. 204

5.1 A Mixed Kleisli Approach with Syntactic Effects ........................................... 204

5.1.1 The Problems with Direct Application of Mathematical Operational Semantics .......................................................... 204

5.1.2 Introducing the Mixed Kleisli Setting ......................................................... 206

5.1.3 Order-Enrichment for a Semantic Domain ............................................... 207

5.1.4 Operational Specifications for Syntactic Effects ....................................... 207

5.1.5 Adequacy and Compositionality for Syntactic Effects ............................... 208
5.2 Introducing Comodels and Semantic Effects ........................................... 209

5.2.1 Operational Specifications ................................................................. 211

5.2.2 Semantic Domains .............................................................................. 211

5.2.3 A Categorical Proof of Adequacy and Compositionality ....................... 212

5.3 Applications, Adaptations and Limitations ............................................. 213

5.3.1 Non-strict Monads ............................................................................. 214

5.3.2 The Number of Steps-to-Termination .................................................. 214

5.3.3 Working in a Category of Algebras ..................................................... 215

5.4 Conclusion .............................................................................................. 216

Bibliography .................................................................................................. 217
Chapter 1

Introduction

1.1 Background

1.1.1 Transition Systems Are Coalgebras

Many systems studied in computer science may be expressed as transition systems of varying complexity, from automata to process algebras and programming languages – in which one considers how the state of the system evolves over time. In doing so, one requires a set of tools for reasoning about the behaviour of these systems – in particular, for deciding whether or not two systems exhibit similar behaviour, and for establishing that they satisfy desirable properties.

In recent decades, the theory of coalgebra has been proposed as a unifying framework for addressing these questions [Rut00]. By expressing the systems under study as a class of $B$-coalgebras $\gamma : X \to BX$ – assigning behaviours $BX$ to states $X$, for some functor $B$ – one has recourse to a standard toolkit for reasoning about the behaviour of states. For instance, one has a proof method given by coalgebraic bisimilarity for deciding whether or not one can distinguish the behaviours of two states $x_1, x_2$.

An example, which we return to later, is given by non-deterministic labelled transition systems (lts’s), where each state $x$ can exhibit a set of labelled transitions $x \xrightarrow{a} x'$ to other states $x'$, or termination $x \xrightarrow{\triangleright}$; such a system may be represented as a $B$-coalgebra for the functor $BX = \mathcal{P}(A \times X + 1)$ – in other words, a function $X \to \mathcal{P}(A \times X + 1)$ (where $\mathcal{P}$ is the powerset functor), assigning to each state $x$ a set of labelled transitions, represented by a subset of $(A \times X + 1)$. In this setting, coalgebraic bisimilarity instantiates to the established notion of strong bisimilarity on labelled lts’s.
This criterion of behavioural equivalence is often characterised by a final coalgebra $D$, in the following sense. Intuitively, its elements describe all the possible transition behaviours of states; and for each coalgebra $\gamma : X \to BX$, one has a canonical map (coalgebra morphism) $\beta_\gamma : X \to D$ from its carrier $X$ into the final coalgebra $D$, mapping each state $x$ into a description of its transition behaviour. Under mild conditions, any two states are coalgebraically bisimilar (i.e. have the ‘same’ behaviour) if and only if they are sent to the same element of the final coalgebra by these maps.

However, in the study of transition systems, there are often a variety of behavioural equivalences of interest, other than coalgebraic bisimilarity. In the context of labelled lts’s, strong bisimilarity is at the fine-grained end of this spectrum; on the opposite, coarse-grained end is trace equivalence. It has been shown [HJS07, PT99] that under certain conditions, this equivalence may also be described in terms of a final coalgebra, this time in a Kleisli category $\text{Kl}(M)$ for a relevant monad $M$. In the case of labelled lts’s, the transition functor $BX = \mathcal{P}(A \times X + 1)$ contains the powerset functor $\mathcal{P}$, which is a monad; and one may show that a final coalgebra in $\text{Kl}(\mathcal{P})$ allows us to assign to each coalgebra state $x$ its collections of finite-length traces $x \xrightarrow{a_1} \ldots \xrightarrow{a_n} x_n \xrightarrow{\nu}$. Thus, two states $x, y$ are identified by the maps into the final coalgebra if and only if they have the same finite-length traces.

### 1.1.2 Process Algebra and Programming Language Semantics

In the mid-nineties, the applications of coalgebra to automata theory were extended by Turi and Plotkin to a wide class of process algebras in terms of coalgebras, in a framework which they called mathematical operational semantics [Tur96, Kli07].

In specifying such languages, one typically begins by defining the syntax terms $T_0$ of the language, and then one provides an operational specification $\epsilon$ of how these terms would execute, or behave, on a hypothetical computer. Such specifications are often given syntactically by operational rules, and each specification induces an implementation, or operational model $\text{om}$, of the language.

To prove properties about program behaviour in $T_0$, it is useful to have a notion of operational equivalence $\cong$, for deciding whether or not two programs $p, q$ are considered to have the same behaviour, written $p \cong q$. An alternative perspective is offered by denotational semantics, in which one reasons about programs $p$ mathematically by assigning them suitable interpretations $[p]$, or denotations, in some semantic domain.

One often wishes the denotational semantics to be compositional, meaning that we can determine the denotations of programs in a modular way, given the denotations of their parts. This means that for each syntax constructor $\sigma$ of the language, we have a corresponding interpre-
1.1. Background

This denotational semantics should also be adequate with respect to the operational semantics: if two programs $p, q$ are denotationally or ‘mathematically’ equivalent – i.e. $[p] = [q]$ – then their behaviours should be indistinguishable: $p \cong q$.

To ensure the properties of adequacy and compositionality, one often restricts the operational specifications of the language into a particular form, or congruence format; and one must then prove that these properties are satisfied by such specifications. Such proofs are typically long and tedious, and must be done on a case-by-case basis for different process algebras.

Turi and Plotkin gave an elegant, coalgebraic solution to this problem. In their framework, one represents the syntax constructors of the language by a functor $\Sigma$, and transition behaviours by a functor $B$. The language is viewed as a transition system, where the states are given by the terms $T0$ of the language (the initial $\Sigma$-algebra); and their executions are described by the transitions of an operational model, represented by a suitable coalgebra $om : T0 \rightarrow BT0$.

Operational specifications may be formalised in different ways as distributive laws of syntax over behaviour. One possible way is as follows: given the behaviours of the arguments $x_1, \ldots, x_n$ of a syntax term $\sigma(x_1, \ldots, x_n)$, an operational specification should tell us the resulting behaviour of that term. One may express this kind of specification abstractly as a natural transformation $\epsilon$, called an abstract operational specification in [TP97]. These distributive laws instantiate to different rule formats for different classes of transition systems, by changing the behaviour functor $B$. For instance, in the case of non-deterministic labelled transition systems (lts’s), one recovers the GSOS congruence format. Each distributive law, equivalent to a specification of the language, induces some operational model $om$ by an inductive process called structural recursion.

As described above for transition systems, the final coalgebra $D$ now gives rise to a notion of behavioural equivalence $\cong$ for programs, where $p \cong q$ whenever $p$ and $q$ they are identified via the final-coalgebra morphism $\beta_{om} : T0 \rightarrow D$. Turi and Plotkin showed that one also obtains a denotational semantics $[-] : T0 \rightarrow D$, by providing interpretations $[\sigma]$ of syntax constructors $\sigma$ on the final coalgebra $D$ – formally, by giving it a $\Sigma$-algebra structure $dm : \Sigma D \rightarrow D$. They then produced a concise proof of adequacy and compositionality of these semantics, by introducing the concept of bialgebras: coalgebras $X \rightarrow BX$ whose state-space also has an algebraic structure $\Sigma X \rightarrow X$. By varying the functors $\Sigma$ and $B$, one is automatically guaranteed adequate and compositional semantics for a wide class of process algebras, including timed systems [Kic04] and probabilistic processes [Bar04].
1.1.3 Notions of Computation and Effects

After the work on mathematical operational semantics, there was an interest in whether or not the theory could be extended to give a coalgebraic semantics to programming languages, in addition to process algebras: for instance, [PP01] concludes with a remark that “one would wish to reconcile this work with the coalgebraic treatment of operational semantics...”

A natural starting point for such a coalgebraic treatment is given by notions of computation, introduced in [Mog91] to provide a unified treatment of many phenomena occurring in programming languages, such as non-determinism, program variables, and interactive user input/output. The key idea was to represent a computation of type $X \rightarrow Y$ as a function $f : X \rightarrow MY$, for a suitable monad $M$ representing a notion of computation. For instance, a non-deterministic computation $X \rightarrow Y$ (in $\text{Set}$) may be represented by a function $f : X \rightarrow \mathcal{P}Y$ mapping each value $x \in X$ into a subset of $Y$, indicating the possible values $f(x)$ may take. In this way, the object $MY$ may be thought of as a collection of ‘computational behaviours’, assigned to elements of the object $X$; if one takes $X = Y$, then one obtains an $M$-coalgebra $f : X \rightarrow MX$, so that the computation $f$ represents a coalgebraic transition system.

Subsequent research refined the understanding of notions of computations, by expressing them in terms of algebraic theories, given by a collection of syntactic operators, called effects, and equations. For many monads, the objects $MX$ can be represented as free models of these theories, with generators given by $X$. One example is given by the algebraic theory of global store, which gives rise to the side-effect monad $MX = (X \times S)^S$ used to describe the semantics of state-modifying programs; here, $S = \mathbb{N}^L$ is the collection of stores, or assignments of values $\mathbb{N}$ to a given, finite set of variables $L$. One may thus view a computation $X \rightarrow Y$, or equivalently a function $X \rightarrow MY$, as mapping each input $x$ to (the equivalence class of) an algebraic term $t(y_1, y_2, \ldots)$ in the free model $MY$ of the algebraic theory, with generators given by $Y$.

These algebraic theories are represented abstractly by Lawvere theories $\mathcal{L}$; this allows us to consider models of algebraic theories in settings other than the category $\text{Set}$ of sets, such as categories of $\omega$-complete partial orders. Moreover, there are canonical constructions such as the tensor and commutative sum of Lawvere theories, allowing us to combine the operations and equations (or sketches) of different theories $\mathcal{L}_1, \mathcal{L}_2$, in contrast to the difficulties of combining notions of computations, expressed by monads.

In addition to the research on models of algebraic effects, it was noted [PS04] that Lawvere theories also gave rise to stateful ‘implementations’ of effects, formally given by comodels. As an example, the comodels of the algebraic theory of global store were shown to correspond to variable arrays; the collection of stores $S = \mathbb{N}^L$ is an example of such a comodel, which responds in the natural way to a request $x = n$ to update variables – mapping a store $s$ to the store $s[x \mapsto n]$ where $x$ has been updated to $n$ – or lookups on the values of variables $x$, which
return the value $s(x)$ of $x$ in the store. However, comodels also provide a notion of state for other effects, such as interactive user I/O and monoid actions [PP08].

This comodel-based view of state may be used to provide operational semantics for effectful computations, as described in [PP08]. One pairs a comodel $C$ of an effect-theory with a model $M$, in a construction called a tensor; we may view the effects of the model $M$ as abstract 'requests', to be passed to the comodel $C$ to carry out their implementations (such as reading and writing to the store). This gives a semantic description of typical stateful transitions such as $(\langle x = n; x = m \rangle, s) \rightarrow (\langle x = m \rangle, s[x \mapsto n])$. This insight raised the question of whether one may also give operational specifications for stateful programming languages, incorporating both notions of effects and comodels.

## 1.2 Goals of this Thesis

As outlined above, the theory of mathematical operational semantics was successful in providing a unified framework for describing the semantics of process algebras; and this led researchers to speculate on its relevance to the semantics of programming languages, and in particular its relationship with the theory of algebraic effects and comodels. The aim of this thesis is to explore the possibilities and limitations of the mathematical operational semantics framework for specifying and reasoning about the behaviours of programming languages in a generic way, and whether this semantics may be expressed in terms of effects and comodels.

These goals may be broken down further into the following components:

- A formal description of syntax, and coalgebraic behaviour, for languages with effects and/or comodels;
- A means of providing operational specifications for these languages;
- A semantic domain for programming languages, with induced behavioural equivalences and denotational semantics;
- A congruence format for operational specifications, and a guarantee that this format ensures adequacy and compositionality of our semantics for programs.

Our starting point for this analysis is the simple **While** language; it incorporates a notion of stateful computation, which may be described in terms of effects – the theory of global store – and comodels, which amount to implementations of the store. We also consider related languages, given by extending the state-free fragment **SWhile** (without variable lookup and update) with these and other effects, such as non-determinism. Moreover, we define a language
NDWhile which extends While with non-determinism, to introduce a more elaborate interaction between comodels and effects. However, our aim is also to preserve the generality of the original framework, by expressing the theory on the same abstract level.

When trying to describe the behaviour of programs by coalgebras, the relevance of Kleisli categories quickly becomes apparent; indeed, it is noted in [Mog91] that the natural way to compose computations \( f : X \to MY \) and \( g : Y \to MZ \) is in the Kleisli category for the monad \( M \). Moreover, as we observe in Sections 3.1.3 and 3.3.2, the work on coalgebraic trace semantics in a Kleisli category is readily adapted to describe the behaviour of such programs. Hence, we take the theory of coalgebras in the Kleisli category as a starting point in our goal of applying mathematical operational semantics to programming languages.

We also considered the related, alternative approach of considering coalgebras in the category of Eilenberg-Moore algebras \( EM(M) \) for the monad \( M \); while this has some advantages over the Kleisli approach, it also raises subtle difficulties, to which it is hard to provide satisfactory solutions. For this reason, our work has focused on coalgebras in a Kleisli setting. We discuss the advantages and limitations of an Eilenberg-Moore setting in Section 5.3.3.

1.3 Thesis Contributions and Structure

The structure of this thesis is as follows:

- We introduce the preliminaries in Chapter 2, covering mathematical operational semantics, coalgebras in a Kleisli category, and computational effects, emphasising the aspects which play a role in the following chapters.

- Then in Chapter 3, we combine these ideas to produce an adaptation of mathematical operational semantics, geared towards programming languages with syntactic effects: in these languages, whenever an effect occurs (such as a variable lookup), it introduces branches in the execution (e.g. depending on the value of the variable in the store); and to keep the development concrete, we do not assume any equations on these effects.

- In Chapter 4, we introduce algebraic theories for these effects, described formally by Lawvere theories, so that we may describe program execution in terms of the monadic notions of computation they induce. Moreover, we give a coalgebraic description of stateful programs, where the state is given by a comodel of a Lawvere theory. We then extend our adaptation of mathematical operational semantics to cover these classes of languages.

- In Chapter 5, we review the contributions of this thesis, and the limitations of our framework; we discuss some possible directions for future work, including the challenges of adapting the framework to an Eilenberg-Moore setting.
In more detail, we outline these contributions below.

- We demonstrate the two main difficulties in applying mathematical operational semantics to programming languages directly, with reference to the While language – represented in terms of $MB$-coalgebras, where $MX = (X \times S)^S$ is the side-effect monad and $BX = V + X$ is a transition function, as described above.

  1. the abstract operational specifications $\epsilon$ instantiate to a rule format which bears little resemblance to the conventional specifications of stateful languages; and we argue that to address this problem, one must be explicit about information which is normally left implicit in these specifications.

  2. The final $MB$-coalgebra $D$ is an excessively fine-grained semantic domain for these languages, distinguishing the behaviours of programs which should be considered equivalent. This is because it records information about state-manipulations at every execution step, rather than the overall effect of a program on the state.

- To address problem (2), we show how a final coalgebra $\overline{D}$ in the Kleisli category $\text{Kl}(M)$ gives rise to a more coarse-grained behavioural equivalence, which is more suitable for characterising the behaviour of these programs; for this reason, we propose a modification of the mathematical operational semantics framework which models syntax in the underlying category, and behaviour in the Kleisli category. To keep the ensuing development concrete, we initially restrict attention to programming languages with syntactic effects, described by $TB$-coalgebras where $T_e$ is a monad freely generating effect-syntax.

- We exploit the methods of existing research in coalgebraic trace semantics, to show how a suitable order-enrichment ensures existence of a final coalgebra $\overline{D}$ in the Kleisli category; and we consider the implications of this order structure for semantic domains, and for program syntax.

- We begin to address problem (1) above, by showing that one may provide an operational specification $\epsilon$ for an effectful language, by extending a simpler specification $\rho$ for the effect-free fragment of the language – such as the fragment $S\text{While}$ of While without variable lookup or update. In this extension process, we make explicit reference to the implicit information described earlier, through the notion of dependency functions $\text{dep}$.

- We outline the behavioural equivalence and denotational semantics induced by the coarse-grained semantic domain, and show that to preserve the key properties of adequacy and compositionality, one must impose a further restriction on the operational specifications introduced above. For this purpose, we introduce the effectfully extended Evaluation-in-Context (eEIC) congruence format. We then reduce the problem of proving adequacy and compositionality to the commutativity of a large diagram, and prove that it commutes by a syntactic argument.
• We generalise the above development away from languages with syntactic effects, by introducing languages whose behaviour is described by monadic notions of computation (written \( N_e \)), as given by free models of Lawvere theories; their operational models are represented by \( N_e B \)-coalgebras. We also define an adaptation \( N_c \) of the side-effect monad, so that we may represent languages with a notion of state given by a comodel, as \( N_c B \)-coalgebras. Lastly, we define monads \( T_{ce}, N_{ce} \) to describe a class of languages incorporating both comodels and effects. We formalise the idea of ‘passing’ effects between a program and a comodel, allowing us to translate from an effect-based to a comodel-based view.

• For each of these classes of languages, we give both concrete (syntactic) and abstract (categorical) formulations of the Evaluation-in-Context rule format (EIC1-3), which are more transparent than the general operational specifications referred to under problem (1) above.

• We show that the semantics of these classes of languages may be described once more by final coalgebras in suitable Kleisli categories. We describe the induced behavioural equivalences on programs, and give examples to illustrate the corresponding denotational models. Finally, we prove adequacy and compositionality of the resulting semantics by appealing to the same large diagram as before, but now adapting our earlier syntactic argument into a fully categorical proof.

Many of the ideas in this thesis were published in our papers [ASP11, ASP13]; however, we give a more extensive treatment of the interactions between models and comodels in Sections 4.1.3 and 4.1.5, and we have adapted the results in [ASP13] to a multi-sorted setting. We also describe effectful languages in an order-theoretical setting from the outset in Chapter 3, improving on the \( \text{Set} \)-based formulation in [ASP11].

1.4 Related Work

The original mathematical operational semantics framework was laid out in [Tur96]; in particular, our strategy for proving adequacy and compositionality is inspired by [RT94], by showing that a final-coalgebra morphism \( \beta_{om} \) is also a \( \Sigma \)-algebra morphism.

A key part of the framework – the derivation of operational models from operational specifications, by structural recursion – was expressed on a more abstract level in [LPW04]; for instance, the ‘abstract operational specifications’ \( \epsilon \) may be seen as instances of distributive laws of a monad over a (cofree) copointed endofunction; and these in turn may be seen as a refinement of distributive laws of monads over comonads [LPW00]. However, we could not see a natural way to define analogous distributive laws for the effectful behaviours we have
considered. Hence, in the following development, we have taken the more concrete approach of inducing operational models by structural recursion, as described in [TP97].

Regarding applications of the framework, in addition to non-deterministic lts’s, it was applied to other classes of process algebras in [Bar04, Kic04, Tur97], and to recursive definitions in an order-theoretical setting in [Kli04]. The case studies in [Tur97] demonstrate the use of categorical (co)products for providing operational specifications. In particular, the example of semilattices with bottom might suggest that this technique is possible in a category of algebras – but that example concerns a commutative monad; and one finds that a key technique in these specifications, of distributing products over coproducts, is unlikely to exist for algebras of non-commutative monads, which includes most monads of interest in program semantics. We return to this point in Section 5.3.3.

The possibilities of final coalgebra semantics in a Kleisli category are explored in [HJS07, Jac04, KK11, PT99]; these introduce variants of the technique employed in Chapter 3, of exploiting initial-algebra/final-coalgebra correspondences in order-enriched categories. (Co)cones over the initial and final sequences in Kleisli categories, which play a part in our proofs of adequacy and compositionality, are described in [HJS07]. In [Jac04], it is observed that possibly-infinite trace semantics gives rise to a weakly final coalgebra; a similar phenomenon occurs in our setting, if one tries to abstract away the number of steps-to-termination, as discussed in Section 5.3.2.

Other ways of inducing final coalgebras in Kleisli categories (and Eilenberg-Moore categories) are discussed in [JSS12, BK10], mostly in the setting of automata theory; but these techniques either apply only to commutative monads, or to classes of behaviour functors $B$ which do not include the natural choice $BX = V + X$ for describing the transition behaviour of programs. More general conditions for the existence of final coalgebras, expressed in terms of locally presentable and accessible categories, are given in [PW98].

Notions of computation are introduced by Moggi in [Mog91]. Computational effects and Lawvere theories are reviewed in [HP07, PP04]; a concrete perspective on algebraic theories in computer science is offered by [Rob02]. The theories of global store, giving rise to the side-effect monad, and local store, are introduced in [PP02]. Countable, discrete, and enriched Lawvere theories are introduced in [Pow06, HP06, Pow99] respectively. Operational rules for languages with effects are considered in [JSV10, PP01]; [PP01] gives a syntactic proof of adequacy, and [JSV10] is based on the idea of evaluation-in-context in an effectful setting, which inspired our congruence formats. Comodels, in particular for global store, are described in [PS04]; and the interaction between models and comodels is discussed in [PP08].
Chapter 2

Background and Preliminaries

2.1 Mathematical Operational Semantics

We begin by introducing the semantic framework of Turi and Plotkin, outlined in [TP97] and described in more detail in [Tur96].

A broad summary of this section is as follows. One defines a programming language, or process algebra, by three components: its syntax constructors, which define the syntax terms of the language; an appropriate notion of transition behaviour, described by coalgebras; and an operational specification, assigning transition behaviour to syntax terms.

The framework unifies two perspectives on program semantics. The first is operational, whereby one studies programs in terms of their behaviour characterised by a final coalgebra. The other is denotational, whereby programs – given by an initial algebra – are assigned meanings via an algebra homomorphism.

In this chapter, we will mostly be working with the category Set of sets and functions between them; but we will also introduce categories of (co)algebras and (co)models, and the category Cpo_{⊥} of ω-cpos.

2.1.1 Syntax and Algebras

In this section, we show how functors may be used to construct program syntax with reference to a simple programming language, which we call BPA for Basic Process Algebra (adapted from [TP97]). In this non-deterministic language, programs p simply ‘output’ a sequence of labels a, b, . . . before stopping. Program syntax is built from three main components:
1. A command \texttt{nil} which has no transitions;

2. Given any program \( p \), a \textit{prefix} of that program \( a.p \) which outputs the label \( a \) and then executes \( p \);

3. Given any two programs \( p, q \), their \textit{parallel composition} \( p \parallel q \) which ‘evaluates them in parallel’ – i.e. it continually chooses either the left or the right argument, and executes that argument for one step.

Example terms include \texttt{nil}, \texttt{a.nil}, \texttt{nil || nil}, and \texttt{a.nil || b.nil}.

Here is the formal definition of this language syntax:

\textbf{Definition 2.1.1.} Given a set of output labels \( A \), the language \textbf{BPA} has syntax terms given by:

\[
P ::= \texttt{nil} \mid a.P \mid P \parallel P
\]

where \( a \) ranges over the set \( A \).

In other words, the syntax of \textbf{BPA} can be described by a constant term \texttt{nil}, a collection of \textit{unary syntax constructors} \( a.(\cdot) \) taking one argument, and a \textit{binary syntax constructor} \( (\cdot) \parallel (\cdot) \) taking two.

In \textbf{BPA}, executing a program \( p \) proceeds in a series of atomic steps, each of which outputs some label \( a \) in \( A \), and gives rise to new program states \( p' \) which must be further evaluated; we will express a typical atomic transition with the notation \( p \overset{a}{\rightarrow} p' \). Thus, some typical transitions we would see in \textbf{BPA} are as follows. Parallel compositions \( \parallel \) may execute non-deterministically in one of several ways, as shown.

\[
\begin{align*}
a.b.c.nil & \overset{a}{\rightarrow} b.c.nil \overset{b}{\rightarrow} c.nil \overset{c}{\rightarrow} \texttt{nil} \\
\texttt{nil || b.nil} & \overset{a}{\rightarrow} \texttt{nil || b.nil} \overset{b}{\rightarrow} \texttt{nil || nil} \\
\texttt{a.nil || b.nil} & \overset{b}{\rightarrow} \texttt{a.nil || nil} \overset{a}{\rightarrow} \texttt{nil || nil}
\end{align*}
\]

\textbf{Functorial Syntax}

Given a set of variables \( X \), we write \( \Sigma X \) for the terms obtained by applying each of the above syntax constructors to the variables \( X \). For instance, if \( X = \{x\} \) and \( A = \{a,b\} \), then \( \Sigma X \) will consist of the terms \( \{\texttt{nil, a.x, b.x, x \parallel x}\} \). and \( \Sigma^2 X \) will contain terms with two layers of syntax constructors:

\[
\begin{align*}
\texttt{nil, a.a.x, a.b.x, b.a.x, b.b.x, a.(x \parallel x), b.(x \parallel x)} \\
\texttt{nil \parallel \texttt{nil}, \texttt{nil \parallel a.x, a.x \parallel \texttt{nil}, \texttt{nil \parallel (x \parallel x), a.x \parallel \texttt{nil, a.x \parallel b.x, etc.}}}
\end{align*}
\]
Suppose we have a function \( f : X \to Y \) – which may be thought of as relabelling the variables in \( X \) into \( Y \)'s. Given a syntax term containing variables in \( \Sigma X \), we may relabel its arguments by applying this function, to obtain a term containing variables in \( Y \). For instance, suppose \( X = \{x, y\} \), \( Y = \{u, v\} \), and \( f : X \to Y \) maps \( x \) to \( u \) and \( y \) to \( v \). Then relabelling the term \( x \parallel a.y \) gives the result \( u \parallel a.v \). We call this term-relabelling function \( \Sigma f \), of type \( \Sigma X \to \Sigma Y \).

It is easy to check that \( \Sigma \) defines an (endo)functor on \( \text{Set} \).

Such syntax functors \( \Sigma \) may be described systematically as polynomial functors. For BPA syntax, we may formally represent \( \Sigma X \) – the terms given by applying syntax constructors to a set of variables \( X \) once – as follows:

- A binary syntax term like \( x \parallel y \) corresponds formally to a pair \((x, y)\) in the cartesian product \( X^2 = X \times X \).
- A unary term \( a.x \) corresponds to an element \((a, x)\) of \( A \times X \). This may be thought of as a collection of multiple ‘copies’ of \( X \), one for every element of \( A \). More abstractly, it is an \( A \)-fold coproduct, or copower, of \( X \); we will make use of copowers when we work in categorical settings other than \( \text{Set} \) in the following chapters.
- For convenience, the constant term \( \text{nil} \) is represented by the unique element \(*\) of the singleton set \( \{*\} \), which we write \( 1 \). If we had multiple such constants \( t_1, t_2, \ldots \) indexed by a set \( V \), they could similarly be represented by the set \( V \).

Taking the disjoint union or coproduct \( + \) of these sets, for BPA syntax we may thus define

\[
\Sigma X = X^2 + A \times X + 1.
\]

To be explicit about which component of a coproduct \( X + Y \) an element \( z \) lies in, we write \( \text{inl}(z) \) (‘in-left’) if \( z \) comes from \( X \), and \( \text{inr}(z) \) (‘in-right’) if it comes from \( Y \). This gives rise to injection functions \( \text{inl}_X : X \to X + Y \) and \( \text{inr}_Y : Y \to X + Y \), also given by the categorical definition. However, for readability we often omit occurrences of \( \text{inl} \) and \( \text{inr} \) for elements \( z \) of \( X + Y \); and we occasionally omit the subscripts \( X, Y \). (As is standard, given two maps \( f : X \to Z \), \( g : Y \to Z \), we write \([f, g] : X + Y \to Z\) for the corresponding function defined on the coproduct \( X + Y \), and similarly for maps \( m : Z \to X \) and \( n : Z \to Y \), we write \( \langle m, n \rangle : Z \to (X \times Y) \) for the product function, defined in \( \text{Set} \) by \( \langle m, n \rangle(z) = (m(z), n(z)) \)).

In addition, given a ‘variable-relabelling’ function or arrow \( f : X \to Y \), it extends naturally to relabel the terms in \( \Sigma X \) using the standard categorical definitions of (co)products. For instance, in \( \text{Set} \) we have the following functions:

- \( f^2 : X^2 \to Y^2 \) for the function which applies \( f \) componentwise to pairs \((x, x')\) in \( X^2 \);
2.1. Mathematical Operational Semantics

- \(A \times f : A \times X \to A \times Y\) applies \(f\) to the \(X\)-component \(x\) of a pair \((a, x)\) in \(A \times X\);

- In overloaded notation, we write ‘1’ for the identity function \(\text{id}_1 : 1 \to 1\). (The previous function \(A \times f\) arises by similarly overloading \(A\) as \(\text{id}_A : A \to A\).)

By applying these functions to each component of \(\Sigma X\), we may now capture the action of \(f\) on terms, via the function

\[
\Sigma f = f^2 + A \times f + 1 : X^2 + A \times X + 1 \to Y^2 + A \times Y + 1.
\]

These definitions generalise as follows. The action of an \(n\)-ary syntax constructor on a set \(X\) may be represented by the \(n\)-fold product functor \(X^n\) – with corresponding ‘relabelling’ function \(f^n : X^n \to Y^n\), given a function \(f : X \to Y\). (Note that this includes constants like \(\text{nil}\), where \(n = 0\) implies \(X^0 = 1\) and \(f^0 = \text{id}_1\).) If there are many of these \(n\)-ary constructors, indexed by a set \(A_n\), for convenience we may represent them collectively by the product functor \(A_n \times X^n\). Lastly, we collect syntax constructors of different arities by taking coproducts \(+\) of the corresponding functors.

**Definition 2.1.2.** Let \(C\) be a category with countable products and coproducts. An endofunctor \(\Sigma\) on \(C\) is **polynomial** if it is of the form \(\Sigma X = \bigsqcup_{n < \omega} A_n \times X^n\) for some \(\mathbb{N}\)-indexed collection of objects \(A_n\).

Indeed, the BPA syntax functor \(\Sigma X = X^2 + A \times X + 1\) takes this form.

However, a syntax functor \(\Sigma\) only describes one application of the syntax constructors. We may construct arbitrary terms of the language by starting with the empty set \(\emptyset\), written \(0\), and ‘repeatedly applying \(\Sigma\’). We illustrate this for BPA (where for simplicity, we assume there is just one label \(a\), so that \(A = \{a\}\)):

- \(0 = \emptyset\)
- \(\Sigma 0 = \{\text{nil}\}\)
- \(\Sigma^2 0 = \{\text{nil}, a.\text{nil}, \text{nil} || \text{nil}\}\)
- \(\Sigma^3 0 = \{\text{nil}, a.\text{nil}, \text{nil} || \text{nil}, a.a.\text{nil}, a.\text{nil} || a.\text{nil}\}\)

Further applications of \(\Sigma\) will result in sets including all the above terms, in addition to deeper syntax terms such as \(a.a.a.\text{nil} \in \Sigma^4 0\).

We will write \(T \Sigma 0\), or more simply \(T 0\), for the set of all syntax terms of the language constructed by \(\Sigma\). Intuitively, it is given by combining all the elements of \(\Sigma^n 0\) for each \(n\) in a consistent manner – i.e. making sure that we identify repeated occurrences of the same term in different sets \(\Sigma^n 0\) (such as \(\text{nil}\) above, which occurs in \(\Sigma 0, \Sigma^2 0,\) and so on).
Remark 2.1.3. Categorically, $T_0$ is given by the colimit of the initial sequence for $\Sigma$, shown below up to ordinal $\omega$; see [Tur96] Chapter 1 for more details. (With the exception of the finite power-set monad, we will never need to consider the sequence beyond the ordinal $\omega$.)

$$0 \xrightarrow{?_{\Sigma}} \Sigma_0 \xrightarrow{\Sigma_0} \Sigma_0 \xrightarrow{?_{\Sigma}} \Sigma_0 \xrightarrow{\Sigma_0} \Sigma_0 \xrightarrow{?_{\Sigma}} \Sigma_0 \xrightarrow{\Sigma_0} \cdots$$

We give some concrete details about what this colimit means. We have written $?_{\Sigma_0}$ for the unique function $0 \rightarrow \Sigma_0$ from the empty set into $\Sigma_0$; informally, it serves to inject one more layer of syntax (in particular, constants) into the $n$-depth terms $\Sigma^n_0$, giving the $(n+1)$-depth terms $\Sigma^{n+1}_0$. One requirement for $T_0$ to be a colimit is that there is a cocone of this diagram with vertex $T_0$, meaning a family of functions $i_n : \Sigma^n_0 \rightarrow T_0$ shown below.

$$0 \xrightarrow{?_{\Sigma}} \Sigma_0 \xrightarrow{i_0} \Sigma_0 \xrightarrow{i_1} \Sigma_0 \xrightarrow{i_2} \Sigma_0 \xrightarrow{i_3} \cdots$$

These include the terms in $\Sigma^n_0$ among the terms in $T_0$ in a consistent manner, meaning that the triangles commute:

$$i_n \circ \Sigma^n_0 = i_n$$

Being a colimit means that $T_0$ is essentially the ‘smallest’ set with this property: for any other cocone – a set $S$ with inclusions $i_n' : \Sigma^n_0 \rightarrow S$ satisfying (†) – there is a unique function $f : T_0 \rightarrow S$ embedding $T_0$ into $S$, such that $i_n' \circ f = i_n$ for all $n$.

To ensure existence of the colimit $T_0$, one commonly assumes that the category has colimits of $\omega$-chains, which is indeed the case in $\textbf{Set}$.

Remark 2.1.4. Concreteness and terms. We have assumed the categorical setting is concrete, allowing us to refer to elements of objects; this will be the case throughout this thesis. In particular, we may refer to individual syntax terms of a language as elements of $T_0$. We omit the concrete details of how the colimit is constructed in $\textbf{Set}$; when we work in $\textbf{Cpo}$ in later chapters, the limit-colimit coincidence spares us from addressing this question, as it allows us to verify the structure of the (co)limits of interest by a least-fixpoint calculation (Remark 3.3.8).

Denotational Semantics via Algebras

In denotational semantics, we characterise the essential properties of a programming language by assigning each program $p$ a mathematical meaning $\llbracket p \rrbracket$ in a semantic domain $D$. This assignment should be compositional, in the sense that the meaning of a program can be determined
by deconstructing the program into its sub-parts, and finding their denotations. These may then be suitably combined to reconstruct the denotation of the original program.

More formally, the previous section showed how a program $p$ may be considered as a syntax-term $\sigma(p_1, \ldots, p_n)$, where $\sigma$ is a syntax constructor (such as parallel composition $\parallel$ or prefixes $a.$) applied to sub-terms $p_i$ (such as $p_1 \parallel p_2$ or $a.p_1$). Compositionality then asserts that the denotation of $p$, or $\sigma(p_1, \ldots, p_n)$, may be constructed from the denotations of its parts $[p_i]$, for some function $[\sigma]$:

$$[\sigma(p_1, \ldots, p_n)] = [\sigma]([p_1], \ldots, [p_n]) \quad \text{(Comp)}$$

This amounts to an assertion that the function $[-]$ is a $(\Sigma)$-algebra homomorphism from terms $T_0$ into denotations $D$. The syntax terms $T_0$ are algebraic, in the sense that we may apply any syntax constructor $\sigma$ to arguments $p_i$, building a new term $\sigma(p_1, \ldots, p_n)$; similarly, given functions $[\sigma]$, denotations may also be considered algebraic, in that we may apply each function $[\sigma]$ to denotations $d_i$, resulting in a new denotation $[\sigma](d_1, \ldots, d_n)$. The syntax constructors of the language become algebraic operators on these sets; thus, compositionality (Comp) asserts that the mapping $[-]$ from programs to denotations respects this algebraic structure.

These ideas are expressed categorically by $\Sigma$-algebras.

**Definition 2.1.5.** For an endofunctor $\Sigma$ on a category $\mathcal{C}$, a $\Sigma$-algebra is a pair $(X, \beta)$ of an object $X$ of $\mathcal{C}$, and an arrow $\gamma : \Sigma X \to X$, its $\Sigma$-algebra structure.

**Example 2.1.6.** The language BPA has syntax functor given by $\Sigma X = X \times X + A \times X + 1$. Thus, a $\Sigma$-algebra consists of a carrier set $D$, and a function $\gamma : D \times D + A \times D + 1 \to D$. It assigns an element of the carrier $D$ to each element of its domain – which consists of a coproduct, or disjoint union, of three sets: the set of pairs $(d, d')$ in $D \times D$; the set of pairs $(a, d)$ in $A \times D$; and a singleton set $1 = \{\ast\}$. Thus, $\gamma$ may be thought of as specifying three functions $\gamma_1, \gamma_2, \gamma_3$, by restricting it to each component of the coproduct. The first, $\gamma_1 : D \times D \to D$, given a pair $(d, d')$, provides an interpretation of parallel composition, $d \parallel d'$, acting on elements of $D$. The second, $\gamma_2 : A \times D \to D$, given a pair $(a, d)$, interprets the prefix $a.d$. Finally, $\gamma_3 : 1 \to D$ simply picks out an element of $D$, interpreting the nil command.

**Example 2.1.7.** A concrete example of a $\Sigma$-algebra – a possible denotational model for BPA – is given by $D = \mathcal{P}(A^*)$. Each element of $D$ is a collection of finite traces $t = a_1 a_2 \ldots a_n$ over $A$, representing the possible sequence of output labels a program may exhibit. In Section 2.2.3, we will show how this semantic domain arises through a final coalgebra in a Kleisli category.

We write $\epsilon$ for the empty trace (where $n = 0$). For instance, the behaviour of nil, having no output activity, would correspond to a single trace $\{\epsilon\}$, $a.b$.nil would correspond to $\{ab\}$, and $a.nil \parallel b.nil$ to the pair of traces $\{ab, ba\}$. (This ‘correspondence’ is in fact the denotational map $[-] : T_0 \to D$ given by the unique $\Sigma$-algebra morphism, as described shortly.)
As for the required algebra structure \( \gamma : \Sigma D \to D \), we may interpret the syntax constructors \( \text{nil}, a.(\cdot), (\cdot) \parallel (\cdot) \) of BPA on these sets of traces in a natural way. The interpretation of \( \text{nil} \) is the empty trace \( \{\epsilon\} \) – i.e. we take \( \gamma_3(*) = \{\epsilon\} \). Given a collection of traces \( d = \{t_i : i \in I\} \), the interpretation of the prefix \( a.d \) is given by prefixing all the traces in \( d \): \( \gamma_2(d) = \{at_i : i \in I\} \). Lastly, parallel composition of \( d = \{s_i : i \in I\} \) and \( d' = \{t_j : j \in J\} \) is given by all interleavings of traces \( s_i \) and \( t_j \). We omit the formal definition; but as an illustration, the traces \( aa \) and \( bb \) have six interleavings: \( \{aabb, abab, abba, baab, baba, bbaa\} \).

Syntax terms \( T0 \) have a canonical \( \Sigma \)-algebra structure, which we call \( \psi_0 \): applying a syntax constructor \( \sigma \) to terms \( p_i \) simply builds a new term \( \sigma(p_1, \ldots, p_n) \). Moreover, this algebra structure is an isomorphism; its inverse \( \psi_0^{-1} : T0 \to \Sigma T0 \) essentially ‘unravels’ the first syntax layer \( \sigma \) of terms \( \sigma(p_1, \ldots, p_n) \) in \( T0 \), to give a syntax constructor applied to sub-terms \( \Sigma T0 \).

**Remark 2.1.8.** Having defined \( T0 \) as a colimit (Remark 2.1.3), this canonical \( \Sigma \)-algebra structure \( \psi_0 \) may be constructed by assuming that \( \Sigma \) preserves colimits of \( \omega \)-chains; and polynomial functors in \( \text{Set} \) satisfy this property. Under this assumption, we may apply \( \Sigma \) to the initial sequence, giving a new sequence:

\[
\Sigma 0 \xrightarrow{\Sigma^0} \Sigma 1 \xrightarrow{\Sigma 1} \Sigma 2 \xrightarrow{\Sigma 2} \Sigma 3 \xrightarrow{\Sigma 3} \Sigma 4 \xrightarrow{\Sigma 4} \cdots
\]

As \( \Sigma \) preserves colimits, \( \Sigma T0 \) is the colimit of this new diagram. However, this diagram is a sub-diagram of the initial sequence, which is easily extended back into the initial sequence (by adding the initial object \( 0 \)); there is a 1-1 correspondence between cocones/colimits of this sub-diagram, and those of the initial sequence. Hence \( \Sigma T0 \) is also a colimit of the initial sequence (as well as \( T0 \)), so that there is a coherent isomorphism \( \psi_0 : \Sigma T0 \cong T0 \); this defines the canonical \( \Sigma \)-algebra structure of terms \( T0 \).

Given a denotational model, such as the \( \Sigma \)-algebra of the previous example, compositionality (Comp) requires that the map \([-\] from programs into denotations is compatible with their algebraic structure. This idea is formalised by saying that \([-\] is a \( \Sigma \)-algebra morphism.

**Definition 2.1.9.** A \( (\Sigma-) \)algebra morphism between two \( \Sigma \)-algebras \((X, \gamma)\) and \((Y, \delta)\) is an arrow \( f : X \to Y \) such that the below diagram commutes.

\[
\begin{array}{ccc}
\Sigma X & \xrightarrow{\Sigma f} & \Sigma Y \\
\downarrow \gamma & & \downarrow \delta \\
X & \xrightarrow{f} & Y
\end{array}
\]
2.1. Mathematical Operational Semantics

It is easy to check that the class of $\Sigma$-algebras forms a category $\text{Alg}(\Sigma)$, with arrows given by $\Sigma$-algebra morphisms.

Thus, for a semantic domain $D$, we seek a compositional denotational semantics: a $\Sigma$-algebra morphism $[-] : T_0 \to D$. This may be achieved straightforwardly by induction, once we are given an algebra structure for $D$ – i.e. a function $\gamma : \Sigma D \to D$, encoding the interpretations of syntax constructors $[\sigma]$ on denotations. For instance, suppose we wish to interpret the syntax term $(a.a.\text{nil} \parallel b.b.\text{nil}) \in T_0$ in the semantic domain $D = \mathcal{P}(A^*)$ of traces given by the previous example. We would proceed as follows:

- $[\text{nil}] = \{\epsilon\}$ by definition; $\gamma_1(\text{nil}) = \{\epsilon\}$
- $[a.\text{nil}] = \{a\}$ prefixing $\epsilon$ with $a$ gives the trace $a$
- $[a.a.\text{nil}] = \{aa\}$ and $[b.b.\text{nil}] = \{bb\}$ similarly
- $[a.a.\text{nil} \parallel b.b.\text{nil}] = \{aabb, abab, abba, baab, baba, bbba\}$ Interleaving interpretation of $\parallel$

Formally, we obtain this inductive assignment by exploiting the fact that $T_0$ – the colimit of the initial sequence for $\Sigma$ – is the initial $\Sigma$-algebra; this means that for any $\Sigma$-algebra $D$, there is a unique $\Sigma$-algebra morphism $[-] : T_0 \to D$ from $T_0$ into $D$, which we may take to be an assignment of denotations $d \in D$ to programs $p \in T_0$.

Remark 2.1.10. Under the assumptions of existence of $\omega$-colimits and $\omega$-continuity of $\Sigma$, to show that $T_0$ is the initial $\Sigma$-algebra, we note that every $\Sigma$-algebra $(X, \gamma)$ induces a cocone over the initial sequence (see [Tur96] pp.35). Thus there is a unique arrow $T_0 \to X$ mediating between the cocone structures of $T_0$ and $X$. Every such arrow may be shown to be a $\Sigma$-algebra morphism $T_0 \to X$, and vice-versa; hence there is exactly one such algebra morphism, implying that $T_0$ is the initial algebra.

2.1.2 The Free Syntax Functor $T$

Given a functor $\Sigma$ corresponding to the syntax constructors of a language, the assumptions of Remarks 2.1.3 to 2.1.10 guarantee the existence of an initial $\Sigma$-algebra $T_0$, which represents the syntax terms of the language. This construction may be generalised: given a collection of constants $Y$, we may build the collection of syntax terms over $Y$, containing constants $y \in Y$ in addition to the syntax constructors. To achieve this effect, one may add these constants to the syntax definition of the language; for instance, we may extend BPA syntax with these constants, as follows:

$$P ::= \text{nil} \mid a.P \mid P \parallel P \mid y \in Y.$$  

This corresponds to replacing the syntax functor $\Sigma$ with $\Sigma + Y$. 
Example 2.1.11. In the case of BPA, the functor \((\Sigma + Y)\) applied to \(X\) gives

\[(\Sigma + Y)(X) = \Sigma(X) + Y = X^2 + A \times X + 1 + Y.\]

As before, this may be interpreted as an application of the syntax constructors of BPA to syntax variables \(x \in X\); but in addition to the parallel compositions \(x \parallel x'\), input prefixes \(a.x\), and \(\text{nil}\) symbol, the last component \(Y\) of the coproduct introduces constants \(y \in Y\).

This functor maps a ‘relabelling’ function \(f : X \to Z\) into the function

\[(\Sigma + Y)(f) = \Sigma(f) + Y = f^2 + A \times f + 1 + Y\]

where, as before, we overload the symbol \(Y\) to represent the identity function \(\text{id}_Y : Y \to Y\). Relabelling does not affect the new constants \(Y\).

Thus, to obtain arbitrary-depth syntax terms containing constants in \(Y\), we may construct the initial \((\Sigma + Y)\)-algebra in the same way as we did for the initial \(\Sigma\)-algebra. We will call its carrier \(TY\) – the set of syntax terms over \(Y\) – and (in this section only) its structure \(\theta_Y : \Sigma TY \to TY\).

As noted in [Tur96], a \((\Sigma + Y)\)-algebra with carrier \(X\) is a function \(\gamma : \Sigma X + Y \to X\). We may consider \(\gamma\) as two functions defined separately on each component of the coproduct; the first is a \(\Sigma\)-algebra structure \(\gamma_1 : \Sigma X \to X\), giving an interpretation of the syntax constructors on the carrier (as in Example 2.1.6). The other component \(\gamma_2 : Y \to X\) interprets the constants \(Y\) as elements of the carrier \(X\). Thus, a \((\Sigma + Y)\)-algebra with carrier \(X\) may be described by these two functions:

\[\Sigma X \xrightarrow{\gamma_1} X \leftarrow^\gamma_2 Y \quad (\dagger)\]

In particular, as \(TY\) is the initial \((\Sigma + Y)\)-algebra, we will use special symbols \(\psi_Y : \Sigma TY \to TY\) for its \(\Sigma\)-algebra structure (the left arrow above) and \(\eta_Y : Y \to TY\) for the inclusion of the constants \(y \in Y\) as trivial terms in \(TY\) (the right arrow).

We may similarly decompose a \((\Sigma + Y)\)-algebra morphism \(f\) between two such algebras \((X, \gamma)\) and \((Z, \delta)\). It is equivalent to a function \(f : X \to Z\) making the following diagram commute:

\[
\begin{array}{c}
\Sigma X \xrightarrow{\Sigma f} \Sigma Z \\
\gamma_1 \downarrow \quad \delta_1 \\
X \xrightarrow{f} Z \\
\gamma_2 \downarrow \quad \delta_2 \\
Y
\end{array}
\]

Thus, \(f\) is an ordinary \(\Sigma\)-algebra morphism between the \(\Sigma\)-algebra structures given by \(\gamma_1\) and \(\delta_1\), and it also maps the interpretations of constants \(Y\) in \(X\) into their interpretations in \(Z\):
2.1. Mathematical Operational Semantics

\[ f \circ \gamma_2 = \delta_2. \]

We now show how \( T \) may be made into a functor. Concretely, given a function \( f : Y \to Z \), we can relabel occurrences of constants \( y \in Y \) in any term over \( Y \), replacing them with constants \( f(z) \in Z \), and giving a term over \( Z \). This will give the required function \( Tf : TY \to TZ \). For instance, if \( Y = \{y, y'\} \) and \( Z = \{z, z'\} \), then the function \( f : y \mapsto z, y' \mapsto z' \) may be used to relabel the constants in the term \( y \parallel a.y' \), giving the term \( z \parallel a.z' \).

Categorically, the function \( Tf : TY \to TZ \) may be constructed using the fact that \( TY \) is the initial \((\Sigma + Y)\)-algebra. It is sufficient to make \( TZ \) into a \((\Sigma + Y)\)-algebra; then initiality guarantees a unique \((\Sigma + Y)\)-algebra morphism \( TY \to TZ \), which we may take to be the definition of \( Tf \). To achieve this, we need to give \( TZ \) a \( \Sigma \)-algebra structure, and also provide a function \( Y \to TZ \) interpreting the constants \( Y \) in \( Z \). This is straightforward: \( TZ \) already has a \( \Sigma \)-algebra structure \( \psi_Z \), and we may use the function \( f : Y \to Z \), composed with \( \eta_Z : Z \to TZ \), for the other requirement. Thus, \( Tf \) is defined to be the unique \((\Sigma + Y)\)-algebra morphism making the below diagram commute:

\[
\begin{array}{c}
\Sigma TY & \xrightarrow{\Sigma Tf} & \Sigma TZ \\
\psi_Y & \downarrow & \psi_Z \\
TY & \xrightarrow{Tf} & TZ \\
\eta_Y & \downarrow & \eta_Z \\
Y & \xrightarrow{f} & Z
\end{array}
\]

We refer to \( T \) as the \textit{free} \( \Sigma \)-algebra functor; given a set of constants \( Y \), it constructs a syntactic \( \Sigma \)-algebra \( \psi_Y : \Sigma TY \to TY \) generated from those constants. A key property of this kind of free construction is what is known as \textit{inductive extension} in [Tur96]. Suppose we are given a \( \Sigma \)-algebra \( A = (A_0, \gamma : \Sigma A_0 \to A_0) \), and a function \( f : Y \to A_0 \) which interprets the generators \( Y \) of the free algebra \( TY \) as elements of the carrier \( A_0 \). One may extend the interpretation to arbitrary terms \( f^* : TY \to A_0 \), where syntax constructors are interpreted by the algebra structure \( \gamma \). The constants \( Y \), interpreted as elements of \( A_0 \) by \( f \), serve as ‘base cases’ for this inductive extension.

Formally, the inductive extension is obtained as follows. Given a \( \Sigma \)-algebra \( A = (A_0, \gamma) \) and a function \( Y \to A_0 \), we obtain a \((\Sigma + Y)\)-algebra:

\[
\Sigma A_0 \xrightarrow{\gamma} A_0 \xleftarrow{f} Y
\]

As a result, because \( TY \) is the initial \((\Sigma + Y)\)-algebra, we obtain a unique \((\Sigma + Y)\)-algebra
morphism $g$ from $TY$ into $A_0$ making the following diagram commute:

\[
\begin{array}{c}
\Sigma TY \xrightarrow{\Sigma g} \Sigma A_0 \\
\downarrow{\psi_Y} \quad \downarrow{\gamma} \\
TY \xrightarrow{\gamma} A_0 \\
\downarrow{\eta_Y} \\
Y \\
\end{array}
\]

Example 2.1.12. Let $D = \mathcal{P}(A^*)$ be the denotational model for BPA given in Example 2.1.7. Fix an output label $a \in A$ and a number $n \in \mathbb{N}$, and take $Y = \{0, 1, 2, \ldots, n\}$. Let $f : Y \rightarrow D$ be the function mapping each $m$ to the trace $a \ldots a$ of length $m$. The resulting inductive extension $g : TY \rightarrow D$ maps an arbitrary syntax term over $Y$ into an element of $D$ (i.e. a collection of traces). For instance:

\[
\begin{array}{ccc}
g : & 3 & \mapsto \{aaa\} \\
& b.3 & \mapsto \{baaa\} \\
& b.0 & \mapsto \{b\} \\
& 3 \parallel b.0 & \mapsto \{aaab, aaba, abaa, baaa\}
\end{array}
\]

as $f(3) = \{aaa\}$

interpretation of prefixing $b$ given by $D$

similarly $(f(0) = \{\epsilon\})$

interpretation of parallel composition in $D$

2.1.3 Free Constructions, Adjunctions, and Monads

As with other free constructions, $T$ may be associated with an adjunction. We write $U^\Sigma$, or simply $U$, for the forgetful functor $\text{Alg}(\Sigma) \rightarrow \text{Set}$, which maps a $\Sigma$-algebra $A = (A_0, \gamma)$ to its underlying carrier $UA = A_0$ – ‘forgetting’ the algebra structure $\gamma$ – and which trivially maps an algebra morphism $f : X \rightarrow Y$ to its underlying arrow $f$ in Set.

Conversely, we write $F^\Sigma$, or just $F$, for the free $\Sigma$-algebra functor $\text{Set} \rightarrow \text{Alg}(\Sigma)$ mapping a set $X$ to the $\Sigma$-algebra $TX$ given by the set of terms over $X$, with structure $\psi_X : \Sigma TX \rightarrow TX$; and arrows $f : X \rightarrow Y$ are mapped to the corresponding algebra morphisms $Tf : TX \rightarrow TY$ produced by the functor $T$.

Inductive extensions give rise to an adjunction $F \dashv U$, i.e. a natural isomorphism between two homfunctors $\text{Set}(X, UA) \cong \text{Alg}(\Sigma)(FX, A)$. The left-to-right component of the isomorphism is as follows. Given a $\Sigma$-algebra $A = (A_0, \gamma)$, its carrier $A_0$ is by definition $UA$; so, any map $f : X \rightarrow A_0 = UA$ may be thought of as providing interpretations of the symbols $X$ in the algebra $A$. Its inductive extension $f^*$ is then a $\Sigma$-algebra morphism $f^* : TX \rightarrow UA$; seen as an arrow in the category of algebras $\text{Alg}(\Sigma)$, $f^*$ is an arrow of type $FX \rightarrow A$, from the free $\Sigma$-algebra $FX = (TX, \psi_X : \Sigma TX \rightarrow TX)$ into the algebra $A$. (The right-to-left component is similar.)
We may now express $T$ as the composition $U F$; in other words, free syntax may be generated by producing a free $\Sigma$-algebra, and forgetting its algebra structure. The unit of this adjunction $\eta_X : X \to UFX$ is the second component of the $(\Sigma + X)$-algebra structure of $TX$ in equation (‡), of type $X \to TX$.

We will make use of the fact that the adjunction makes $T$ into a monad. For future reference, we frame monads in concrete terms: informally, they may be thought of as functors $M$ which build a collection of structures $MY$ containing occurrences of elements of $Y$. This includes 'trivial' containers for single elements of $Y$, constructed by the unit, $\eta_Y : Y \to MY$. Moreover, nested structures $M^2Y$ over $Y$ may be combined into a single layer of structure $MY$ by the multiplication $\mu_Y : M^2Y \to MY$.

**Definition 2.1.13.** A monad $M$ on a category $C$ is an endofunctor with two natural transformations $\eta_X : X \to MX$ and $\mu_X : M^2X \to MX$ – the unit and multiplication respectively – making the following diagrams commute:

\[
\begin{array}{ccc}
MX & \xrightarrow{M\eta_X} & M^2X \\
\downarrow{\eta_M} & & \downarrow{\mu_X} \\
M^2X & \xrightarrow{\mu_X} & MX
\end{array}
\quad
\begin{array}{ccc}
M^3 & \xrightarrow{M^2\mu_X} & M^4 \\
\downarrow{\mu_M} & & \downarrow{\mu_X} \\
M^2X & \xrightarrow{\mu_X} & MX
\end{array}
\]

We will often be working with several monads at the same time; to avoid confusion, we sometimes decorate the unit $\eta_X^M : X \to MX$ and multiplication $\mu_X^M : M^2X \to MX$ to indicate to which monad they belong.

In the case of the syntactic monad $T$, one may interpret the unit $\eta$ and multiplication $\mu$ as follows. Given a set of constants $X$, the map $\eta_X : X \to TX$ treats the constants as trivial syntax terms in $TX$. As for the multiplication, applying $T$ to $X$ twice gives the set of *syntax terms over syntax terms* over $X$. The multiplication $\mu_X : T^2X \to TX$ ignores the distinction between the two layers of syntax, producing terms in $TX$.

### 2.1.4 Monadic Strength and Enrichment

Throughout this thesis, we will frequently assume that monads $M$ have a *strength*. In $\text{Set}$, this is given by a natural transformation $\text{st}_{X,Y} : X \times MY \to M(X \times Y)$. Intuitively, it allows us to ‘pair up’ an element $x$ of $X$ with each occurrence of an element $y_i$ in the structures $MY$, giving a new structure $M(X \times Y)$ containing pairs $(x, y_i)$. For instance, in the case of syntax terms $t(y_1, y_2, \ldots)$ in $TY$, this amounts to pairing up a given $x$ in $X$ with each argument of the term, giving a new term $t((x, y_1), (x, y_2), \ldots)$ in $T(X \times Y)$. There is also an analogous notion of *costrength*, $\text{cost}_{X,Y} : MX \times Y \to M(X \times Y)$, which instead attaches an occurrence of $y$
to structures in \( M \times X \). In more generality, the definition of strength is given with respect to a **monoidal product** \( \otimes \) rather than the categorical product \( \times \).

**Definition 2.1.14.** A **monoidal product** on a category \( C \) is a (bi)functor \( \otimes : C \times C \to C \), with a **unit object** \( I \) and natural isomorphisms \( \alpha \) (‘associator’) and \( \beta, \gamma \) (‘left and right unitor’) of type

\[
\alpha_{A,B,C} : A \otimes (B \otimes C) \cong (A \otimes B) \otimes C \\
\beta_A : I \otimes A \cong A \\
\gamma_A : A \otimes I \cong A
\]

satisfying two coherency conditions ([Kel05] diagrams 1.1 and 1.2). The product is also **symmetric** if there is a natural transformation \( \delta_{A,B} : A \otimes B \cong B \otimes A \) satisfying three coherency conditions ([Kel05] diagrams 1.14 to 1.16).

**Definition 2.1.15.** A monad \( M \) on a category \( C \) with a monoidal product \( \otimes \) is **\((\otimes)-strong\)** if it has a **strength** – a natural transformation \( st_{X,Y} : X \otimes MY \to M(X \otimes Y) \) satisfying four coherency conditions ([Mog91] Definition 3.2). The corresponding **costrength** is a natural transformation \( cost_{X,Y} : MX \otimes Y \to M(X \otimes Y) \) satisfying four analogous conditions.

If \( \otimes \) is symmetric, one may define the costrength in terms of the strength as follows:

\[
\begin{align*}
\text{cost}_{X,Y} : MX \otimes Y &\xrightarrow{\delta_{MX,Y}} Y \otimes MX \\
&\xrightarrow{\text{st}_{X,Y}} M(Y \otimes X) \\
&\xrightarrow{M \delta_{Y,X}} M(X \otimes Y).
\end{align*}
\]

We may guarantee monadic strength by assuming additional structure on the category \( C \):

**Definition 2.1.16.** A category \( C \) is **symmetric monoidal** if it has a symmetric monoidal product \( \otimes \); it is also **closed** if for each object \( A \) of \( C \), the functor \( X \mapsto A \otimes X \) (and hence, also \( X \mapsto X \otimes A \)) has a right adjoint, \( X \mapsto X^A \).

We may consider symmetric monoidal closed categories \( C \) as being **enriched** over themselves ([Kel05] Section 1.6). We will consider this enrichment in the case \( C = \text{Cpo}_{\perp} \) in more detail in Section 2.2.3 (Definition 2.2.12); but the point for us is that in such categories, strong monads are equivalent to \( C \)-enriched monads, as argued in Remark 3.5 in [Mog91]. Hence, to show that a monad is strong, it is sufficient to show it has a \( C \)-enrichment. We will do this for a syntactic monad \( T_e \) in Section 3.3.3 (Lemma 3.3.11), and for monads generated by a Lawvere theory in Section 4.3.2.

### 2.1.5 Distributivity

We will also make frequent use of a property of closed categories: they are **distributive** with respect to the monoidal product \( \otimes \). (In a symmetric setting, we need not concern ourselves with
2.1. Mathematical Operational Semantics

This means that there is a natural isomorphism \( \text{dist} \) as shown below, with the given inverse (making use of injections \( \text{inl}_X : X \to X+Y \) and \( \text{inr}_Y : Y \to X+Y \)).

\[
\text{dist}_{X,Y,Z} : (X + Y) \otimes Z \longrightarrow (X \otimes Z) + (Y \otimes Z)
\]

\[
(\text{dist}^{-1})_{X,Y,Z} = [\text{inl}_X \otimes \text{id}_Z, \text{inr}_Y \otimes \text{id}_Z] : (X \otimes Z) + (Y \otimes Z) \longrightarrow (X + Y) \otimes Z
\]

One may show that there is such a natural transformation \( \text{dist} \), with the given inverse, by exploiting the fact that the functor \( P \mapsto (P \otimes Z) \) is a left adjoint, hence it preserves colimits, and in particular the coproduct \(+\). This means that the object \((X+Y) \otimes Z\), with the injection arrows \( \text{inl}_X \otimes \text{id}_Z \) and \( \text{inr}_Y \otimes \text{id}_Z \), satisfies the universal property of the coproduct of \( X \otimes Z \) and \( Y \otimes Z \). Hence, there is a unique arrow \( \text{dist}_{X,Y,Z} \) making the top two triangles commute in the following diagram. To show that its inverse is \( [\text{inl}_X \otimes \text{id}_Z, \text{inr}_Y \otimes \text{id}_Z] \), note that the bottom two triangles commute by the categorical definition of this map, and hence the central vertical path is a cocone morphism from \((X+Y) \otimes Z\) to itself; this arrow is unique by universality, and the identity arrow is certainly a cocone morphism, so they must coincide: \( [\text{inl}_X \otimes \text{id}_Z, \text{inr}_Y \otimes \text{id}_Z] \circ \text{dist}_{X,Y,Z} = \text{id} \).

A similar argument holds for the composition in reverse order, demonstrating that \( (\text{dist})^{-1} \) is as defined above. (We omit the proof that \( \text{dist} \) and its inverse are natural transformations, as it is routine.)

\[\begin{array}{c}
X \otimes Z \\
\text{inl}_X \otimes \text{id}_Z \\
\text{inl}_X \otimes \text{id}_Z \\
\text{inl}_X \otimes \text{id}_Z
\end{array} \quad \begin{array}{c}
\text{(dist)}_{X,Y,Z} \\
\text{inr}_Y \otimes \text{id}_Z \\
\text{inr}_Y \otimes \text{id}_Z
\end{array} \quad \begin{array}{c}
Y \otimes Z \\
\text{inl}_X \otimes \text{id}_Z \\
\text{inr}_Y \otimes \text{id}_Z \\
\text{inr}_Y \otimes \text{id}_Z
\end{array}
\]

\[
\begin{array}{c}
(X + Y) \otimes Z \\
(X \otimes Z) + (Y \otimes Z)
\end{array}
\]

2.1.6 Behaviour and Coalgebras

Now we focus on the problem of representing program execution. The idea is to treat the programming language as an instance of a particular kind of transition system, given by a coalgebra.

We illustrate how this may be achieved for the language \( \text{BPA} \), by considering it as a non-deterministic labelled transition system (lts). We have introduced notation \( p \xrightarrow{a} q \) to express the fact that a \( \text{BPA} \) program \( p \) may transition to a new program \( q \), outputting a label \( a \) in the process; e.g. \( a.\text{nil} \xrightarrow{a} \text{nil} \). We call this an \( a \)-transition. For each label \( a \), the program \( p \) may have any finite number of \( a \)-transitions \( p \xrightarrow{a} q \) to new programs \( q \). We collect these into a set \( Q_a = \{ q : p \xrightarrow{a} q \} \); then the collection of sets \((Q_a)_{a \in A}\) give a full description of the behaviour of \( p \).
Example 2.1.17. Assuming $A = \{a, b, c\}$, the term $p = a.\text{nil} \parallel b.\text{nil} \parallel b.c.\text{nil}$ has three possible transitions:

\[
\begin{align*}
    a.\text{nil} & \parallel b.\text{nil} \parallel b.c.\text{nil} \\
    a.\text{nil} \parallel b.\text{nil} \parallel b.c.\text{nil} & \rightarrow a.\text{nil} \parallel \text{nil} \parallel b.c.\text{nil} \\
    a.\text{nil} \parallel b.\text{nil} \parallel b.c.\text{nil} & \rightarrow a.\text{nil} \parallel b.\text{nil} \parallel c.\text{nil}
\end{align*}
\]

Thus, to $p$ we would assign $Q_a = \{\text{nil} \parallel b.\text{nil} \parallel b.c.\text{nil}\}$, $Q_b = \{a.\text{nil} \parallel \text{nil} \parallel b.a.\text{nil}, a.\text{nil} \parallel b.\text{nil} \parallel c.\text{nil}\}$, and $Q_c$ is the empty set $\{\}$. Each set $Q_a$ is a finite collection of program states in $P$ – i.e. a finite subset of $P$, and an element of $\mathcal{P}_f(P)$. We may encapsulate all this information about the behaviour of program $p$ by a single ‘transition function’ $f_p : a \mapsto Q_a$, which tells us, for each $a$, all the $a$-transitions of $p$. Thus, we may characterise the transition behaviour of each program $p$, by assigning it the corresponding transition function:

$$
\gamma : P \to \mathcal{P}_f(P)^A, \quad p \mapsto f_p
$$

This function gives a complete specification of the transition system, assigning to each state $p$ a description of its behaviour.

To generalise to other kinds of transition systems, we may replace $BP = \mathcal{P}_f(P)^A$ by an arbitrary behaviour functor $B$ applied to $P$. Such a transition system can thus be described as a co\textit{algebra}: a state-space $P$, coupled with a function $\gamma : P \to BP$ assigning behaviours to programs.

Definition 2.1.18. For an endofunctor $B$ on a category $C$, a $B$-coalgebra is a pair $(P, \gamma)$ of an object $P$ of $C$ – its carrier – and an arrow $\gamma : P \to BP$, its $(B$-coalgebra) structure.

We often refer to a coalgebra $(P, \gamma)$ by its structure map $\gamma$, or occasionally just its carrier $P$. Each functor $B$ defines a particular class of transition systems; for instance, coalgebras for the functor $BX = \mathcal{P}_f(X)^A$ are non-deterministic lts’s with labels in $A$.

This allows us to give a coalgebraic treatment of the operational semantics of the language – a description of how programs execute step-by-step – by making the set of syntax terms $T_0$ into an operational model: a $B$-coalgebra $(T_0, \text{om} : T_0 \to BT_0)$, describing the transition behaviour of programs. We will soon show how this may be achieved; but first, we consider the problem of how to compare the behaviour of programs.
2.1. Mathematical Operational Semantics

Behavioural Equivalence

One often wishes to identify similarities between states $p$ and $q$ of transition systems; in particular, one requires a notion of operational or behavioural equivalence to identify whether or not two programs can be distinguished by an external observer. For instance, one could reasonably say the BPA terms $a.\text{nil}$ and $a.\text{nil} \parallel \text{nil}$ have the same behaviour, as they both have only one $a$-transition, in contrast to $a.\text{nil}$ and $a.a.\text{nil}$, which produce different numbers of successive $a$-transitions.

In direct analogy to $\Sigma$-algebras, one may define $B$-coalgebra morphisms $f$ which make a correspondence between the states of two coalgebras, by matching the immediate transitions of one coalgebra’s states $p$ with the transitions of states $f(p)$ in another coalgebra.

**Definition 2.1.19.** A $(B)$-coalgebra morphism between two $B$-coalgebras $(X, \gamma)$ and $(Y, \delta)$ is an arrow $f : X \rightarrow Y$ such that the below diagram commutes.

$$
\begin{array}{c}
X \xrightarrow{f} Y \\
\gamma \downarrow \quad \delta \\
BX \xrightarrow{Bf} BY
\end{array}
$$

As for algebras, one may define a category $\text{Coalg}(B)$ of $B$-coalgebras, where the arrows are given by $B$-coalgebra morphisms.

Behavioural equivalence for coalgebras is often described in terms of an equivalence relation called bisimilarity. We do go into further details here, except to say that it generalises the standard notion of strong bisimilarity for non-deterministic lts’s, to other classes of transition systems. The key point for us is that the behavioural equivalence expressed by bisimilarity may be elegantly characterised in terms of a final coalgebra, if one exists.

**Definition 2.1.20.** A $B$-coalgebra $(Z, \zeta : Z \rightarrow BZ)$ is final if, for every $B$-coalgebra $(X, \gamma : X \rightarrow BX)$, there is a unique $B$-coalgebra morphism $\beta_\gamma$ from $(X, \gamma)$ into $(Z, \delta)$.

**Example 2.1.21.** For non-deterministic lts’s, where $BX = \mathcal{P}_t(X)^A$, the final $B$-coalgebra is given by a quotient of the set of finitely branching trees with $A$-labelled branches ([Tur96] pp.166). It may be thought of as describing all possible execution paths of states in a non-deterministic lts, where the branching describes the non-determinism that may occur at each transition step. For instance, the behaviour of the program $a.b.(c.\text{nil} \parallel d.\text{nil})$ corresponds to the following element of the final coalgebra.

```
  a  b  c  d  e
\downarrow \downarrow \downarrow \downarrow \downarrow
  b  c  d  e
```
The transition structure $\zeta : Z \to BZ$ of the final $B$-coalgebra is obtained naturally by reading off the transitions of these trees. For instance, one has the transition sequence

$$
\bullet \xrightarrow{c} \bullet \xrightarrow{d} \bullet \quad \bullet \xrightarrow{c} \bullet \xrightarrow{d} \bullet \xrightarrow{d} \bullet
$$

and also a similar sequence with $c$ and $d$ reversed.

A final coalgebra serves as a characterisation of all possible behaviours that states $p$ can exhibit; every state $p$ in a transition system $(P, \gamma)$ may be canonically assigned a unique element $z = \beta_\gamma(p)$ of the final coalgebra, which tells us everything that can be observed about its behaviour.

Under mild conditions on the behaviour functor (preservation of weak pullbacks), one may show that two programs $p, q$ are bisimilar precisely when they are identified by the maps into the final coalgebra: $\beta_\gamma(p) = \beta_\delta(q)$. Thus, the final coalgebra plays a central role in reasoning about the behaviour of coalgebras.

Dually to the initial algebra $T0$ describing syntax terms, one may construct the final coalgebra $Z$ as the limit of the final sequence for the functor $B$, shown below (up to the ordinal $\omega$). We write $!_X$ for the unique function $X \to 1$ into the singleton set $\{\ast\}$.

$$
1 \xleftarrow{!_{B1}} B1 \xleftarrow{B1!_{B1}} B^21 \xleftarrow{B2^2!_{B1}} B^31 \xleftarrow{B^3!_{B1}} \ldots
$$

One may consider the sequence as a series of approximants to the observable ‘global’ behaviours of a state. The singleton 1 represents an arbitrary state; $B1$ represents all possible transitions to arbitrary states; $B^21$ indicates sequences of up to two transitions, and so on. (‘Up to two’ refers to the fact that the first transition may terminate.) The arrows $B^n!_{B1} : B^{n+1}1 \to B^n1$ essentially discard the observable information of the final transition of $(n+1)$-length sequences, giving $n$-length sequences. A categorical limit describes all possible transition sequences, both finite (when some transition is terminal) and infinite (when no terminal transition occurs).

**Example 2.1.22.** Consider the behaviour functor $BX = 1 + A \times X$ – whose coalgebras $X \to BX$ are essentially deterministic lts’s, assigning each state $x \in X$ either the value $\text{inl}(\ast)$ (where $\ast$ is the unique element of 1) if $x$ terminates, or the value $\text{inr}(a, x')$ if $x$ has the transition $x \xrightarrow{a} x'$. The final $B$-sequence is equivalent to the following diagram, where we have implicitly used distributivity isomorphisms $X \times (Y + Z) \cong X \times Y + X \times Z$.

$$
1 \xleftarrow{!_{B1}} 1 + A \xleftarrow{B1!_{B1}} 1 + A \times 1 + A^2 \xleftarrow{B^21!_{B1}} 1 + A \times 1 + A^2 \times 1 + A^3 \times 1 \xleftarrow{!_{B1}} \ldots
$$

Each approximant $1 + \ldots + A^n \times 1$ may be thought of as the set of all completed traces of
length $m < n$, and the set of incomplete (non-terminated) traces of length $(n + 1)$. Within each approximant, the occurrence of 1 in the last set $A^n \times 1$ represents the final, non-terminal state in an incomplete trace; whereas the other 1’s in the sets $A^m + 1$ (for $m < n$) represent completed traces. In the limit, one obtains the collection $A^* + A^\omega$ of both finite and infinite $A$-traces.

**Remark 2.1.23.** Dually to the initial sequence (Remark 2.1.8), for any functor $B$, the final $B$-sequence up to ordinal $\omega$ always has a limit in $\text{Set}$ (as it is a complete category). One may show this limit is the final coalgebra, under the assumption that $B$ preserves limits of $\omega^{op}$-chains; this is the case for polynomial functors of finite arities. More generally, for finitary functors – such as the finitary power-set functor $P_f$ – one may continue the final sequence to the ordinal $\omega + \omega$ to show that a final coalgebra arises as a subset, or subobject of that limit [Wor05].

### 2.1.7 Combining Operational and Denotational Semantics via the Final Coalgebra

We have shown how to define syntax for a programming language in terms of a functor $\Sigma$, which gives rise to a notion of denotational semantics: given a semantic domain $D$ expressed as a $\Sigma$-algebra, there is a unique $\Sigma$-algebra morphism $[\cdot] : T_0 \to D$ (as $T_0$ is the initial $\Sigma$-algebra); the fact that it is an algebra morphism means that it assigns denotations to programs in a compositional manner.

Conversely, having chosen a functor $B$ to represent transition behaviours, we aim to express an operational model for the programming language as a $B$-coalgebra $(T_0, \text{om} : T_0 \to BT_0)$, with carrier given by the terms $T_0$ of the language. If it exists, a final coalgebra $\langle Z, \zeta \rangle$ induces a canonical notion of operational equivalence for programs: two programs $p, q$ are considered behaviourally equivalent whenever they are identified by the unique coalgebra morphism $\beta_{\text{om}} : T_0 \to Z$ from the operational model into the final coalgebra. The key question is how to obtain an operational model $\text{om}$, and a semantic domain $D$, which ensure that the denotational semantics is adequate with respect to operational equivalence. This means that if two programs $p, q$ have the same denotation – i.e. $[p] = [q]$ – then they should be behaviourally indistinguishable, in that $\beta_{\text{om}}(p) = \beta_{\text{om}}(q)$.

The approach taken by Turi and Plotkin is to make the carrier $Z$ of the final coalgebra $\langle Z, \zeta \rangle$ into a denotational model, by producing a suitable $\Sigma$-algebra structure $\text{dm} : \Sigma Z \to Z$. Thus, the $\Sigma$-algebra morphism $[-]$ and $B$-coalgebra morphism $\beta_{\text{om}}$ – characterising denotational and operational equivalence respectively – now have the same type, $T_0 \to D$. (We will always take final coalgebras as denotational models, hence from now on we will use the letter $D$ for the carrier of the final coalgebra.)
We can represent the resulting situation by diagram 2.1 below.

\[
\begin{array}{cccccc}
  \Sigma T0 & \stackrel{\Sigma [-]}{\longrightarrow} & \Sigma D \\
  \downarrow & \uparrow & \downarrow & \uparrow \\
  T0 & \stackrel{[\_]}{\longrightarrow} & D \\
  \downarrow & \uparrow & \downarrow & \uparrow \\
  om \downarrow & \beta om & om \downarrow & \uparrow & \beta \downarrow \\
  BT0 & \stackrel{B\beta om}{\longrightarrow} & BD & (2.1)
\end{array}
\]

Adequacy follows if we can prove equality of the central arrows in the above diagram: if two programs have the same denotation, they will also be identified by the morphism into the final coalgebra, hence will be operationally equivalent; furthermore, the denotational assignment will be compositional because it is a $\Sigma$-algebra morphism.

The rest of this section shows how a suitable congruence format, suitably formulated in categorical terms, may be used to obtain both operational and denotational models om, dm, and also ensure adequacy of the resulting semantics.

### 2.1.8 Operational Models From Abstract Operational Semantics (aOS)

One often specifies the atomic transition steps of programs using Structured Operational Semantics (SOS) rules [Plo04].

**Example 2.1.24.** The SOS rules for BPA are as follows.

\[
\begin{align*}
  & x \rightarrow x', y \rightarrow y' \quad x \rightarrow x', y \rightarrow y' \\
  & a.x \xrightarrow{a} x \quad x \parallel y \rightarrow x' \parallel y \quad x \parallel y \rightarrow x \parallel y'
\end{align*}
\]

There are no rules for nil, as it has no transitions. The rule for an output prefix $a.p$ asserts that it produces an $a$-transition to the prefixed term $p$. The multiple rules for parallel composition $x \parallel y$ imply that there is more than one possible transition, depending on the transitions of the sub-terms $x$ and $y$.

We outline how such operational rules may be expressed categorically as natural transformations. For every syntax constructor $\sigma(x_1, \ldots, x_n)$ – with ‘placeholder’ arguments given by variables $x$ in some set $X$ – if we are given the transition behaviour of each argument $x_i$, the rules allow us to deduce the behaviour of the overall term $\sigma(x_1, \ldots, x_n)$.

**Example 2.1.25.** This works as follows for parallel composition $\parallel$ in BPA. Suppose we are given a term $x \parallel y$, where $x, y \in X$, and also the behaviours of $x$ and $y$ – elements $b_x, b_y$ of
2.1. Mathematical Operational Semantics

\[ BX = \mathcal{P}_I(X)^A, \] which we may write as functions, using lambda-notation:

\[ b_x = \lambda a. (P_a) \quad \text{and} \quad b_y = \lambda a. (Q_a) \quad \text{where } P_a, Q_a \subseteq X \text{ for all } a \in A. \]

(Recall that \( P_a = \{ x_i : i \in I_a \} \) encapsulates all the \( a \)-transitions \( x \xrightarrow{a} x_i \) of \( x \), and similarly for \( y \) and \( Q_a = \{ y_j : j \in J_a \} \).

From the information in the functions \( b_x \) and \( b_y \), and the SOS rules above, we may deduce the transition behaviour of \( x \parallel y \). Its \( a \)-transitions occur when either \( x \) or \( y \) has an \( a \)-transition, to \( x_i \) or \( y_j \) respectively; in these cases, \( x \parallel y \) has \( a \)-transitions to \( x_i \parallel y \), or \( x \parallel y_j \), respectively. Thus, we may define the behaviour of \( x \parallel y \) in terms of \( x, y, b_x, \) and \( b_y \), as follows:

\[ b_{x\parallel y} : \lambda a. (\{ x_i : x_i \in b_x(a) \} \cup \{ x \parallel y_j : y_j \in b_y(a) \}) \]

Note that \( b_{x\parallel y} \) describes \( a \)-transitions to terms over \( X \), such as \( x \parallel y_j \); thus it is an element of \( BTX = \mathcal{P}_I(TX)^A \), and not \( BX \) (which only describes transitions to variables in \( X \)).

Another example is given by output prefixing \( a.x \), where one may define

\[ b_{a,x} = \lambda b. \begin{cases} \{ x \} & \text{if } b = a \\ \{ \} & \text{otherwise} \end{cases} \]

Here there is no program syntax on the right-hand side (above, we had \( x_i \parallel y \) and \( x \parallel y_j \)), but only an occurrence of \( x \). This may nonetheless be considered as a ‘base-case’ constant in \( TX \) (e.g. via the map \( \eta_X : X \rightarrow TX \)), so that \( b_{a,x} \) is again of type \( BTX \).

Thus, for each syntax constructor \( \sigma(x_1, \ldots, x_n) \), one may define a behaviour \( b_{\sigma(x_1,\ldots,x_n)} \) depending on the arguments \( x_i \in X \) and their behaviours \( b_{x_i} \in BX \). The result is an element of \( BTX \). For an \( n \)-ary syntax constructor, this amounts to seeking a function \( (X \times BX)^n \rightarrow BTX \). Given such a function for each syntax constructor \( \sigma \), we may take their coproduct to express them as a single function, in terms of the polynomial functor \( \Sigma \):

\[ \epsilon_X : \Sigma(X \times BX) \rightarrow BTX \]

Moreover, this assignment should be natural, in that it should make no essential difference if we relabel the variables \( x_i \mapsto x'_i \) – the resulting behaviour \( b_{\sigma(x'_1,\ldots,x'_n)} \) should correspond to the old one, under the same relabelling. This amounts to requiring that \( \epsilon \) be a natural transformation. Each \( \epsilon \) of this type may be thought of as an abstract operational semantics for the programming language, or ‘abstract OS’ for short.

Such operational specifications may be thought of as distributing syntax over behaviour; there are different formulations of this idea, of varying expressivity and convenience [Bar04, LPW04].
The class of abstract OS specifications arises from *distributive laws of the syntax monad* $T$ *over a co-pointed endofunctor* $H$; when $H$ is co-freely generated by the behaviour functor $B$, such distributive laws are equivalent to natural transformations $\epsilon$ of the form we have given above [LPW04].

For each choice of behaviour functor $B$, the set of abstract OS specifications gives rise to a *congruence format*, describing the operational rule-sets which can be expressed by such natural transformations $\epsilon$. In the case of non-deterministic lts’s, abstract OS may be shown to correspond to the well-known *GSOS format* [TP97, Bar04]. Behaviour functors for *probabilistic non-determinism* and *timed processes* give rise to other formats [Bar04, Kic04].

We briefly mention that there is a *dual* abstract OS format. First, we overload the symbol $D$ (‘denotational model’) to also refer to the cofree comonad generated by an adjunction $U^B \dashv G^B$ induced by cofree $B$-coalgebras, in direct analogy to the monad $T$ and adjunction $F^\Sigma \dashv U^\Sigma$ induced by free $\Sigma$-algebras. (In fact, the final $B$-coalgebra is given by $D1$, just as the initial $\Sigma$-algebra is given by $T0$.) We will rarely refer to the comonad $D$ outside the remainder of this section.

The dual abstract OS format is then given by natural transformations $\varrho : \Sigma DX \rightarrow B(DX + \Sigma DX)$. In the context of non-deterministic lts’s, the dual format corresponds concretely to the congruence format of *safe ntree rules* [TP97].

### Obtaining an Operational Model from an Abstract OS

We now show how an abstract OS specification $\epsilon$ gives rise to both an operational model $om : T0 \rightarrow BT0$, and a denotational model $dm : \Sigma D \rightarrow D$. The most concrete formulation of this process is given by the following categorical version of *structural recursion with accumulators*.

**Proposition 2.1.26.** [TP97] Given an arrow $h : \Sigma(X \times Y) \rightarrow Y$ and an arrow $s : X \rightarrow Y$, there is a unique arrow $! : TX \rightarrow Y$ making the below diagram commute.

\[\begin{array}{ccc}
\Sigma TX & \xrightarrow{\Sigma(id,s)} & \Sigma(TX \times Y) \\
\downarrow{\psi_x} & & \downarrow{h} \\
TX & \xrightarrow{!} & Y \\
\downarrow{n_x} & & \\
X & \xrightarrow{s} & \\
\end{array}\]

This result may be interpreted as follows. Suppose we wish to assign an accumulator value, of type $Y$, to each term $TX$ over variables $X$. To achieve this by structural recursion, one must assign values $Y$ to the ‘base cases’: the variables $X$ themselves. The function $s : X \rightarrow Y$ tells us how to do this. As for the inductive part of the process, suppose we have a syntax
term \( t = \sigma(t_1, \ldots, t_n) \) built from a syntax constructor \( \sigma \) and sub-terms \( t_i \). Inductively, one may suppose that accumulator values \( Y \) for the sub-terms \( t_i \) have already been derived. Then the function \( h \) tells us how to obtain a value \( Y \) for the original term \( t = \sigma(t_1, \ldots, t_n) \), given the terms \( t_i \) and their accumulator values.

Given an abstract OS \( \epsilon \), we may apply this result with the intention of assigning to each term \( TX \) (with variables \( X \)) an accumulator value \( Y = BTX \) given by the term’s behaviour. The base case requires a map \( X \to BTX \); a convenient way to obtain this is to assume we are given a \( B \)-coalgebra structure \( X \to BX \) for the ‘base-case’ variables \( X \), and post-compose this map with \( B \) applied to the unit of the monad \( T \), \( B\eta_X : BX \to BTX \). (This construes the variables in the behaviour \( BX \) as simple terms in \( TX \), giving a behaviour \( BTX \) over terms.)

As for the inductive case, we require a map \( \Sigma(TX \times BTX) \to BTX \), indicating how to derive the behaviour \( BTX \) of a term \( \sigma(t_1, \ldots, t_n) \), given its sub-terms \( TX \) and their behaviours \( BTX \). We may obtain this by taking the \( TX \) component of the abstract OS map \( \epsilon_{TX} \), and post-composing with \( B \) applied to the multiplication of the monad \( T \), \( B\mu_X : T^2X \to TX \):

\[
h : \Sigma(TX \times BTX) \xrightarrow{\epsilon_{TX}} BT^2X \xrightarrow{B\mu_X} BTX.
\]

The previous result then implies existence of a map \( TX \to BTX \) giving a \( B \)-coalgebra structure to syntax terms generated by \( X \). We may do this for any given abstract OS \( \epsilon \) and coalgebra structure \( \gamma : X \to BX \) on the generators: we write \( T^\epsilon(\gamma) \) for the resulting map. In particular, we may take an empty set of generators \( 0 \) with the trivial coalgebra structure \( ?_0 : 0 \to B0 \) to obtain an operational model for the closed terms of the language, given by \( T^\epsilon(\gamma) : T0 \to BT0 \), which we have called \( \text{om} \).

This technique also allows us to make the final \( B \)-coalgebra \( D \) into a denotational model \( \text{dm} : \Sigma D \to D \). Given an abstract OS, we apply structural recursion to syntax terms over denotations \( TD \) with base cases given by the final coalgebra structure \( \zeta : D \to BD \). This yields a transition structure \( T^\epsilon(\zeta) : TZ \to BTZ \) for terms over denotations. Once we have done this, by finality of \( D \) there is a unique coalgebra-morphism \( \beta_\epsilon : TD \to D \) from this coalgebra back into \( D \) – an assignment of ‘denotations’ \( D \) to arbitrary terms \( TD \) over denotations, not just ‘depth-1’ terms \( \Sigma D \) as required. Thus \( \beta_\epsilon \) is more general than \( \text{dm} \), which may be obtained from the following composition (where \( \psi_D \) is the \( \Sigma \)-algebra structure of \( TD \)).

\[
\text{dm} : \Sigma D \xrightarrow{\Sigma \eta_D} \Sigma TD \xrightarrow{\psi_D} TD \xrightarrow{\beta_\epsilon} D
\]

**Example 2.1.27.** In the context of \( \text{BPA} \) and non-deterministic lts’s, consider these elements of the final \( B \)-coalgebra \( D \) (for \( a \in A \), or ‘denotations’):

\[
d_a : \bullet \xrightarrow{a} \bullet \quad d_0 : \bullet
\]
Each denotation $d_a$ has a single $a$-transition into $d_0$, which has no transitions. Now consider $t := d_a \parallel d_b \in TD$ (for $a, b \in A$); structural recursion tells us that it has two transitions: $t \overset{a}{\rightarrow} d_0 \parallel d_b$ and $t \overset{b}{\rightarrow} d_a \parallel d_0$. Similarly, these have further $b$ and $a$-transitions respectively to $d_0 \parallel d_0$ which has no transitions. Thus, the original term $t$, containing denotations $d_a$ and $d_b$, is mapped by $\text{dm}$ to the following denotation in the final coalgebra:

$$d_a \parallel d_b \mapsto \bullet \overset{a}{\rightarrow} \bullet \overset{b}{\rightarrow} \bullet \overset{b}{\rightarrow} \bullet \overset{a}{\rightarrow} \bullet$$

### 2.1.9 Adequacy and Compositionality

Having obtained operational and denotational models $\text{om, dm}$ in terms of the final coalgebra, we are now in the situation represented earlier, by Diagram 2.1 in Section 2.1.7; to prove adequacy and compositionality, it remains to prove that $[-] = \beta_\text{om}$ as argued there.

Turi and Plotkin achieved this by introducing an elegant theory of bialgebras. We may define a bialgebra $(X, \gamma, \delta)$ to be a carrier $X$ with both a “denotational” algebra structure $\gamma : \Sigma X \rightarrow X$, interpreting syntax constructors, and an “operational” coalgebra structure $\delta : X \rightarrow BX$ subject to the following coherence axiom – which essentially asserts compatibility of these structures with the abstract OS specification $\epsilon$. Here, $\gamma^+ : TX \rightarrow X$ is an extension of $\gamma$ to arbitrary-depth terms in $TX$ (not just depth-1 terms in $\Sigma X$): formally, it is the unique $(\Sigma + X)$-algebra morphism from $TX$ into the $(\Sigma + X)$-algebra $\Sigma X \xrightarrow{\gamma} X \xleftarrow{\text{id}} X$.

$$\begin{array}{ccc}
\Sigma X & \xrightarrow{\gamma} & X \\
\downarrow{\Sigma(id,\delta)} & & \downarrow{\delta} \\
\Sigma(X \times BX) & \xrightarrow{\epsilon_X} & BTX
\end{array}$$

A bialgebra morphism between two bialgebras $(X, \gamma, \delta), (Y, \gamma', \delta')$ is an arrow between their carriers $X \rightarrow Y$ that is both a $\Sigma$-algebra morphism and a $B$-coalgebra morphism between the respective structures. This gives rise to a category of bialgebras, $\text{Bialg}$. Diagram 2.1 contains two bialgebras: $b_0 : \Sigma T0 \xrightarrow{\psi_0} T0 \xrightarrow{\text{om}} BT0$, and $b_1 : \Sigma D \xrightarrow{\psi} D \xrightarrow{\zeta} BD$. The key result is that $b_0$ is the initial bialgebra, and $b_1$ the final bialgebra. Modulo the equivalences $\text{Alg}(\Sigma) \cong \text{Alg}(T)$ and $\text{Coalg}(B) \cong \text{Coalg}(D)$, the proof proceeds in several steps, as outlined below. See [TP97] for more details.

1. Abstract OS specifications (and their dual form) are instances of a more general class of transformations: distributive laws $\lambda_X : TDX \rightarrow DTX$ of the monad $T$ over the cofree comonad $D$.

---

1There are several equivalent formulations, depending on whether one uses the functors $\Sigma, B$ or their induced (co)monads $T, D$. We take the former approach.
2. Each such $\lambda$ is shown to correspond to a lifting $T_\lambda$ of the monad $T$ to the category of $B$-coalgebras, $\text{Coalg}(B)$, and also a lifting $D_\lambda$ of the comonad $D$ to the category of $\Sigma$-algebras, $\text{Alg}(\Sigma)$.

3. The (co)algebras for these functors are equivalent to bialgebras, giving these equivalences of categories: $\text{Alg}(T_\lambda) \cong \text{Bialg} \cong \text{Coalg}(D_\lambda)$.

4. Given a distributive law $\lambda$, the adjunctions $F^\Sigma \dashv U^\Sigma$ and $U^B \dashv G^B$ defining the syntactic monad $T$, and the behavioural comonad $D$, are shown to have explicit liftings $\overline{F}^\Sigma \dashv \overline{U}^\Sigma$ and $\overline{U}^B \dashv \overline{G}^B$ to the corresponding categories $\text{Coalg}(D_\lambda)$ and $\text{Alg}(T_\lambda)$ – where the lifted forgetful functor $\overline{U}^\Sigma$ maps a $D_\lambda$-coalgebra to its carrier, a $\Sigma$-algebra; and similarly, $\overline{U}^B$ maps a $T_\lambda$-algebra to its carrier, a $B$-coalgebra.

5. Moreover, applying $\overline{F}^\Sigma$ to the initial $\Sigma$-algebra gives the bialgebra $b_0$ (up to isomorphism), and similarly $\overline{G}^B$ applied to the final $B$-coalgebra gives $b_1$. The adjoint functors $\overline{F}^\Sigma$ and $\overline{G}^B$, being left and right adjoints respectively, preserve colimits and limits respectively. Thus, $b_0$ – the image under $\overline{F}^\Sigma$ of the initial $\Sigma$-algebra – must also be initial in $\text{Coalg}(D_\lambda)$, and hence an initial bialgebra. Similarly, $b_1$ is shown to be a final bialgebra.

By initiality of $b_0$ and finality of $b_1$ as bialgebras, there is a (doubly unique) bialgebra morphism $! : T_0 \to D$ from $b_0$ to $b_1$ which is both a $\Sigma$-algebra and $B$-coalgebra morphism. It must coincide with the denotational assignment $[-]$, as it a $\Sigma$-algebra morphism out of $T_0$, and there is exactly one $\Sigma$-algebra morphism from the initial $\Sigma$-algebra $T_0$ into any other. Similarly, it must also coincide with the coalgebra morphism $\beta_{\text{om}}$ into the final coalgebra. Hence $\beta_{\text{om}} = ! = [-]$, as required to guarantee adequacy and compositionality.

2.2 Trace Semantics via Kleisli Coalgebras

By choosing different behaviour functors $B$, a variety of process algebras may be formulated in the coalgebraic framework of Turi and Plotkin. In all cases, the notion of behavioural equivalence is derived from coalgebraic bisimilarity; as mentioned earlier, this instantiates to strong bisimilarity for non-deterministic lts’s. However, in some contexts, one would prefer a more coarse-grained criterion of behavioural equivalence, such as trace equivalence. For instance, the following transition behaviours have the same traces, whereas they are distinct elements of the final coalgebra $D$ for non-deterministic lts’s.
In attempting to model trace semantics for coalgebras, one approach has received some attention in recent years, which is to describe the transition systems of interest as coalgebras in a Kleisli category [HJS07, PT99, Jac04]. In this section, we review how a Kleisli category induces a more coarse-grained notion of behavioural equivalence than coalgebraic bisimilarity. This will provide the motivation for our decision to describe program behaviour in a Kleisli category in Chapter 3.

### 2.2.1 Kleisli Categories

**Definition 2.2.1.** Let \( M \) be a monad on a category \( C \). The Kleisli category \( \text{Kl}(M) \) for the monad \( M \) has the same objects as \( C \), but its arrows \( f' : X \to Y \) from \( X \) to \( Y \) (‘Kleisli-arrows’) are given by the arrows \( f : X \to MY \) from \( X \) to \( MY \) in the underlying category. We call such an arrow \( f : X \to MY \) in \( C \) the *underlying arrow* of the arrow \( f' : X \to Y \) in \( \text{Kl}(M) \).

(In special cases, for readability we omit the decoration ' we use the same symbol \( f \) for both arrows.)

Arrows compose in the Kleisli category as follows. Given \( f' : X \to Y \) and \( g' : Y \to Z \) in the Kleisli category with underlying arrows \( f, g \), we define \( g \circ f' \) as the arrow in \( \text{Kl}(M) \) corresponding to the following composition in \( C \).

\[
X \xrightarrow{f} MY \xrightarrow{Mg} MMZ \xrightarrow{\mu} MZ.
\]

Given an underlying arrow \( g : Y \to MZ \), the *extension* of \( g \), written \( g^\dagger : MY \to MZ \), is the arrow given by \( \mu_Z \circ Mg \). The above composition is thus equivalent to \( g^\dagger \circ f \).

In the context of transition systems, one may interpret Kleisli composition as follows. An underlying arrow \( f : X \to MY \) may be thought of as assigning to each \( X \) a collection \( MY \) of ‘branches of \( Y \)’s’ (where the nature of the branches depends on the monad \( M \)). To compose this with another arrow \( g : Y \to MZ \), one uses \( g \) to evaluate the \( Y \)’s on each branch; this gives an element of \( MMZ \), describing two layers of branching, where the inner branches contain \( Z \)’s. The monad multiplication connects the two layers of branching, in a way specific to the monad, to produce a collection \( MZ \) of branches containing \( Z \)’s. The arrow \( g^\dagger : MY \to MZ \) may then be thought of as an extension of a function \( g : Y \to MZ \) to arbitrary collections \( MY \).

**Example 2.2.2.** Take the monad \( M = \mathcal{P}_f \) to be the finite power-set monad – so that \( MX \) is the set of finite subsets of \( X \). Suppose \( f : X \to \mathcal{P}_f Y \) maps \( x \) to the set \( \{y_1, y_2\} \). Furthermore, suppose \( g : Y \to \mathcal{P}_f Z \) maps \( y_1 \) to \( \{z_1, z_2\} \) and \( y_2 \) to \( \{z_3\} \). Then the composition \( g^\dagger \circ f' \) acts on \( x \) as shown below; the multiplication of the power-set monad collapses the nested subsets of \( Z \).
by taking their set union.

\[ x \xrightarrow{f} \{y_1, y_2\} \xrightarrow{Mg} \{\{z_1, z_2\}, \{z_3\}\} \xrightarrow{\mu_Z} \{z_1, z_2, z_3\} \]

The function \( g^\dagger \) maps subsets of \( Y \) to subsets of \( Z \), as illustrated by the assignment \( \{y_1, y_2\} \mapsto \{z_1, z_2, z_3\} \) above.

It is easy to embed the underlying category \( C \) inside the Kleisli category, via the following functor \( J \); there is also a functor \( L \) in the converse direction, giving rise to an adjunction \( J \dashv L \).

**Definition 2.2.3.** Given a monad \( M \) on category \( C \), the **inclusion functor** \( J : C \to \text{Kl}(M) \) is the identity-on-objects functor mapping an arrow \( f : X \to Y \) in \( C \) to the composition \( X \xrightarrow{\eta_Y} Y \xrightarrow{\mu_Y} MY \), seen as an arrow \( X \to Y \) in \( \text{Kl}(M) \). Its right adjoint \( L : \text{Kl}(M) \to C \) maps objects \( X \) to \( MX \), and arrows \( f' : X \to Y \) to the corresponding extensions \( f^\dagger : MX \to MY \).

**Remark 2.2.4.** We will make frequent use of the following easily-proven properties of the inclusion functor \( J \), in particular when reasoning about coalgebra morphisms involving the final coalgebra; it will have a structure of the form \( Jg \) for a suitable map \( g \), and we will frequently use the fact, shown below, that \((Jg)^\dagger = Mg\). This is useful when pre-composing the arrow \( Jg \) with another arrow in the Kleisli category.

Given an arrow \( g : X \to Y \) in \( C \), \( Jg \) corresponds to the underlying arrow \( \eta_Y \circ g \). Pre-composing \( Jg \) with an arrow \( f' : W \to X \), given by an arrow \( f : W \to MX \) in \( C \), gives \((\eta_Y \circ g)^\dagger \circ f \) in \( C \). However, we have (by functoriality of \( M \) and the monad laws):

\[ (Jg)^\dagger = (\eta_Y \circ g)^\dagger = \mu_Y \circ M(\eta_Y \circ g) = \mu_Y \circ M\eta_Y \circ Mg = Mg. \]

Thus, the composition \( Jg \circ f' \) is given by \( Mg \circ f \).

In a similar manner, the post-composition \( h' \circ Jg \) with an arrow \( h' : Y \to Z \) has underlying arrow \( h \circ g : X \to MZ \).

### 2.2.2 Coalgebras in a Kleisli Category

As outlined in [HJS07], some common coalgebraic behaviour functors may be expressed as a composition \( MB \) of two functors:

- A **branching component**, given by a monad \( M \), describing some form of multiplicity;
- A **transition component**, given by a functor \( B \), describing the observable behaviour of each branch.
Example 2.2.5. In the case of non-deterministic lts’s, the behaviour of a state is a finite set of transitions to other states, each of which is decorated with a label $a \in A$. Thus, the branching component $M$ is given by the finitary power set monad $\mathcal{P}I$; a ‘collection of branches’ of elements of $X$ is a set of elements of $X$. The transitions $x \xrightarrow{a} x'$ of a state $x$ are given by pairs $(a, x')$ in $A \times X$ of labels and states; but for technical reasons, it will be convenient to generalise slightly and allow explicit termination (written $x \xrightarrow{\cdot} x'$); thus we take $BX = A \times X + 1$, where the 1 describes termination. (So, a state may non-deterministically terminate, in addition to having $a$-transitions.) We may thus express a non-deterministic lts as an $MB$-coalgebra $X \rightarrow \mathcal{P}I(A \times X + 1)$.

Other examples include probabilistic branching, by taking the monad $M$ to be the sub-distribution monad, and graded transitions, via the bag monad [KK11].

The structure $\gamma$ of an $MB$-coalgebra $(X, \gamma)$ has type $X \rightarrow MBX$; this may be considered as an arrow $\gamma' : X \rightarrow BX$ in the Kleisli category. To construe this as a coalgebra in the Kleisli category, we may seek what is referred to as a lifting in [HJS07] and [PT99].

Definition 2.2.6. Given a functor $B$ and monad $M$ on category $C$, a lifting of $B$ to the Kleisli category $\text{Kl}(M)$ is a functor $\overline{B}$ satisfying $JB = B\text{J}$, where $J$ is the inclusion functor $C \rightarrow \text{Kl}(M)$.

In particular, $J$ is identity-on-objects, so that $\overline{B}$ and $B$ act the same way on objects: $\overline{B}X = BX$. This implies that $MB$-coalgebras $(X, \gamma : X \rightarrow MBX)$ are in 1-1 correspondence with $\overline{B}$-coalgebras $(X, \gamma' : X \rightarrow \overline{B}X)$ – a fact we will use repeatedly throughout this thesis. In more detail, $MB$ is a functor on the underlying category $C$, and $\overline{B}$ a functor on the Kleisli category $\text{Kl}(M)$.

Hence, an $MB$-coalgebra consists of an object $X$ of the underlying category $C$, and an arrow $g : X \rightarrow MBX$ in $C$; a $\overline{B}$-coalgebra consists of an object of $\text{Kl}(M)$ and an arrow $X \rightarrow \overline{B}X$ in $\text{Kl}(M)$. However, note that the objects of $C$ are (by definition) identical to the objects of $\text{Kl}(M)$; and the underlying arrows $X \rightarrow MBX$ are in 1-1 correspondence with arrows of type $X \rightarrow BX$ in the Kleisli category (by definition); finally, the latter are equivalent to arrows $X \rightarrow \overline{B}X$ as $B$ and $\overline{B}$ act the same way on objects.

A convenient way to define liftings $\overline{B}$ of functors $B$ is via distributive laws $\lambda_X : BMX \rightarrow MBX$ of the functor $B$ over the monad $M$ [HJS07].

Definition 2.2.7. A distributive law $\lambda$ of the functor $B$ over the monad $M$, is a natural transformation $\lambda_X : BMX \rightarrow MBX$ making the below diagrams commute:

![Distributive Law Diagram](image_url)
Lemma 2.2.8. [HJS07] There is a 1-1 correspondence between distributive laws between liftings $\overline{B}$ of a functor $B$ to the Kleisli category $\text{Kl}(M)$, and distributive laws $\lambda_X : BMX \rightarrow MBX$.

Given a transition $B$ to a collection of branches $MX$, a distributive law tells us how to ‘attach’ or propagate the transition data (from $B$) to each branch, giving a collection $M$ of transitions $BX$. The right-hand diagram indicates that a transition $B$ may be propagated through nested collections $M^2X$ in a coherent manner, and the left-hand diagram ensures good behaviour with respect to trivial collections (e.g. singleton sets), given by the unit of the monad.

Example 2.2.9. For $M = \mathcal{P}$ and $BX = A \times X + 1$, a distributive law is of type

$$\lambda_X : A \times \mathcal{P}(X) + 1 \rightarrow \mathcal{P}(A \times X + 1).$$

The domain indicates either a terminal transition $\text{inr}(\ast)$, given the singleton set 1, or a pair $\text{inr}(a,s) \in A \times \mathcal{P}(X)$ indicating an $a$-transition to a subset $s$ of successor states in $X$. The transition data – namely, termination $\ast$ or the label $a$ – has to be attached to each successor state.

A terminal transition $\ast$ has no successor states, so we map it (via the unit $\eta_1 : 1 \rightarrow \mathcal{P}(1)$) to the singleton set $\{\ast\}$ containing a single terminal transition; for type-correctness, we must also inject this element $\ast$ into the right-component of the coproduct $A \times X + 1$, giving the singleton set $\{\text{inr}(\ast)\}$, where $\text{inr}_1 : 1 \rightarrow A \times X + 1$ is the right-injection.

To handle an $a$-transition $(a,s)$ to a set of successor-states $s$, we use the monadic strength of $\mathcal{P}$, of type $\text{st}_{A,X} : A \times \mathcal{P}(X) \rightarrow \mathcal{P}(A \times X)$. It attaches the label $a$ to each successor state in $s$, giving a collection of pairs $\{(a,x) : x \in s\}$; this is an element of $\mathcal{P}(A \times X)$. To construe this as an element of $\mathcal{P}BX = \mathcal{P}(A \times X + 1)$, for type-correctness we apply the left-injection $\text{inl}_{A \times X} : A \times X \rightarrow A \times X + 1$ to each pair $(a,x)$, giving the set $\{\text{inl}(a,x) : x \in s\}$.

The resulting definition, incorporating appropriate inclusions $\text{inl, inr}$ into the left and right components of coproducts respectively, is as follows:

$$\lambda_X = [\mathcal{P}(\text{inl}_{A \times X}) \circ \text{st}_{A,X}, \mathcal{P}(\text{inr}_1) \circ \eta_1].$$

(By replacing $A \times X$ with the copower $A \cdot X$ as in [PT99], one avoids the need for monadic strength.)

Example 2.2.10. Throughout this thesis, we will make frequent use of the fact that there is a distributive law of the functor $BX = V + X$ over any monad $M$ as follows ([PT99] Proposition 2.2), which by Lemma [HJS07] defines a lifting $\overline{B}$:

$$\lambda_X : V + MX \rightarrow M(V + X) \quad \lambda_X = [\eta^M \circ \text{inl}_V, M\text{inr}_X]$$
Although $MB$-coalgebras coincide with $\mathcal{B}$-coalgebras, the corresponding coalgebra morphisms are quite different. The $\mathcal{B}$-coalgebra morphisms between two coalgebras $(X, \gamma')$ and $(Y, \delta)$ are the arrows $g$ making the upper-left diagram below commute in $\text{Kl}(M)$; in the underlying category, this corresponds to the diagram on the upper-right, which may be further unravelled into the bottom diagram.

\[
\begin{array}{c}
X \xrightarrow{g} Y \\
\downarrow \gamma' \quad \downarrow \delta' \\
\mathcal{B}X \xrightarrow{\pi g} \mathcal{B}Y
\end{array}
\quad
\begin{array}{c}
X \xrightarrow{g} MY \\
\downarrow \gamma \\
\mathcal{B}X \xrightarrow{(\mathcal{B}g)\dagger} M\mathcal{B}Y
\end{array}
\quad
\begin{array}{c}
X \xrightarrow{g} MY \\
\downarrow \gamma \\
\mathcal{B}X \xrightarrow{\mathcal{B}g} \mathcal{B}MY \\
\downarrow M\delta \\
M^2Y \\
\downarrow \mu_Y \\
\mathcal{B}Y
\end{array}
\]

It is easy to show that every $MB$-coalgebra morphism (between two $MB$-coalgebras) is also a $\mathcal{B}$-coalgebra morphism between the same coalgebras, construed as $\mathcal{B}$-coalgebras. In the case of the finite power set monad $M = \mathcal{P}_t$ (with $BX = A \times X + 1$ and lifting $\mathcal{B}$ given by Example 2.2.9), the $\mathcal{B}$-coalgebra morphisms have been characterised as linear bisimulations [PT99]; but more generally, it is difficult to give such a characterisation of $\mathcal{B}$-coalgebra morphisms for other monads.

### 2.2.3 Constructing the Final Kleisli Coalgebra

#### Towards a Final Kleisli Coalgebra

Just as for $B$-coalgebras, $MB$-coalgebraic bisimilarity is induced by the final $MB$-coalgebra, which arises as a limit of the final sequence for $MB$:

\[
1 \xleftarrow{\lambda_{MB1}} MB1 \xleftarrow{MB1\lambda_{MB1}} MBMB1 \xleftarrow{MB1\lambda_{MB}MB1} \ldots
\]

The approximants $(MB)^n 1$ to the final coalgebra describe alternating sequences of branchings $M$ and transitions $B$; the element $\ast \in 1$ plays the role of an arbitrary state (not a terminal value!), so that the elements of $(MB)^n 1$ describe either: complete $m$-step behaviours for $m < n$; or incomplete $n$-step behaviours (i.e. ending in the state $\ast$). Thus, as a characterisation of behaviour, the final $MB$-coalgebra contains a lot of fine-grained information about the branching behaviour at each step (as described in Example 2.1.21).
If it exists, the final $\mathcal{B}$-coalgebra in a Kleisli category induces a notion of behavioural equivalence which generally differs from $MB$-coalgebraic bisimilarity. This may be illustrated by considering the final sequence for $\mathcal{B}$, which first requires a final object in $\text{Kl}(M)$; i.e. an object, which we temporarily call $1'$, such that there is a unique arrow $!_X : X \to 1'$ in $\text{Kl}(M)$ for every $X$. In the underlying category, this means there is a unique arrow $!_X : X \to M(1')$ for every object $X$; which is equivalent to requiring that $M(1') = 1$. (Recall that $\mathcal{C}$ and $\text{Kl}(M)$ have the same objects.)

For a range of monads $M$ on $\text{Set}$, one finds that the natural candidate for such an object $1'$ is the initial object $0$ in the underlying category; this includes the power-set $\mathcal{P}$, sub-distribution and bag monads (and with a minor tweak, the side-effect monad). Thus, we will assume $M0 = 1$: $0$ is the final object in $\text{Kl}(M)$. We will review this assumption shortly.

However, under that assumption, the final $\mathcal{B}$-sequence $0 \leftarrow \mathcal{B}0 \leftarrow \mathcal{B}^20 \cdots$ corresponds to the following sequence in the underlying category (obtained by applying the functor $L : \text{Kl}(M) \to \mathcal{C}$ of Definition 2.2.3).

$$1 = M0 \leftarrow (\mathcal{B}0)^1 \leftarrow MB0 \leftarrow (\mathcal{B}^20)^1 \leftarrow MB^20 \leftarrow (\mathcal{B}^30)^1 \cdots$$

We compare this to the final $B$-sequence (Example 2.1.22) – where each approximant to the limit $B^n1$ may be thought of as ‘collections of complete traces of length $m < n$ or incomplete traces of length $n$ (ending with an arbitrary state *)’. By contrast, here the occurrences of $B^n0$ correspond only to ‘completed traces of length $m < n$’; there are no incomplete traces. The occurrence of a single $M$ indicates that each approximant $MB^n0$ contains collections of such ‘completed’ traces (in contrast to the multiple $M$’s $MBMB \cdots$ in the final $MB$-sequence). This suggests that the final $\mathcal{B}$-coalgebra will contain all collections $\mathcal{P}(A^*)$ of completed traces, which is the domain for ‘trace semantics’ anticipated in Example 2.1.7. We will shortly make this notion precise, in terms of the initial $B$-algebra.

**Existence via Limit-colimit Coincidence**

Having defined a functor $\mathcal{B}$ on the Kleisli category $\text{Kl}(M)$, a common way to ensure the existence of a final $\mathcal{B}$-coalgebra is to exploit a limit-colimit coincidence, usually in order-enriched categories; this idea is described concretely in [HJS07], and more abstractly in [PT99]. These methods impose conditions relating to the idea of algebraic compactness, whereby every (locally continuous) functor has an initial algebra and a final-coalgebra, and they are canonically equivalent. In an order-enriched setting, algebraic compactness gives a concrete definition of the behavioural equivalence induced by the final coalgebra, as a least-fixpoint construction, which
allows us to reason concretely about the behaviour of programs. We outline each method in turn, referring to [HJS07] for more details.

**Definition 2.2.11.** The category $\mathbf{Cpo}_\omega$ has objects given by $\omega$-complete pointed partial orders, or $\omega$-cppos, and arrows given by all $\omega$-continuous functions between them. The category $\mathbf{Cpo}_{\omega+1}$ is similar, with the extra requirement that all arrows preserve the bottom element: $f(\bot) = \bot$.

We give a concrete definition of $\mathbf{Cpo}_\omega$- and $\mathbf{Cpo}_{\omega+1}$-enrichment, both of categories and functors; we will make frequent reference to $\mathbf{Cpo}$-enrichment in this thesis.

**Definition 2.2.12.** A category $\mathcal{C}$ is $\mathbf{Cpo}_\omega$-enriched – equivalently, $\mathcal{C}$ is a $\mathbf{Cpo}_\omega$-category – if, for all objects $X$ and $Y$, the collection of arrows $f : X \to Y$ forms an $\omega$-cppo – with a bottom element we call $\bot_{X,Y} : X \to Y$, or sometimes just $\bot$ – such that composition of arrows $g \circ f$ is continuous in both arguments: if $f = \sqcup_{n<\omega}(f_n)$ and $g = \sqcup_{n<\omega}(g_n)$, then $\sqcup_{n<\omega}(g_n \circ f) = \sqcup_{n<\omega}(g_n) \circ f$ and $\sqcup_{n<\omega}(g \circ f_n) = g \circ \sqcup_{n<\omega}(f_n)$.

A $\mathbf{Cpo}_\omega$-category $\mathcal{C}$ has left-strict composition if, for all arrows $g : X \to Y$, $\bot_{Y,Z} \circ g = \bot_{X,Z}$. It has right-strict composition if for all $g : Y \to Z$, we have $g \circ \bot_{X,Y} = \bot_{X,Z}$. The category is $\mathbf{Cpo}_{\omega+1}$-enriched if it has strict composition – i.e. both left- and right-strict composition.

It may be shown that left-strictness of composition implies $M0 = 1$ ([HJS07] Lemma 3.5), as we already assumed above. This fact allows us to check if a monad will ensure that arrow composition is left-strict in the Kleisli category.

**Example 2.2.13.** The Kleisli category $\mathbf{Kl}(\mathcal{P})$ is $\mathbf{Cpo}_{\omega+1}$-enriched under a pointwise ordering of the underlying functions $f : X \to \mathcal{P}Y$. This means that for two subsets $s_1, s_2$ of $Y$, we define $s_1 \subseteq s_2$ iff $s_1 \subseteq s_2$. Then $f_1 \subseteq f_2$ iff for every $x \in X$, $f_1(x) \subseteq f_2(x)$. The bottom arrow $\bot_{X,Y} : X \to \mathcal{P}Y$ maps every $x \in X$ to the empty set.

It is easy to check that composition with these bottom arrows is then left- and right-strict, because it produces collections of empty sets, whose union is an empty set. For instance, let $X = \{x\}$, $Y = \{y_1, y_2\}$, and $f : X \to Y$ be the functions with $f(x) = \{y_1, y_2\}$. Then post-composing $f$ with $\bot_{Y,Z} : y_1 \mapsto \{\}$ in the Kleisli category gives

\[
(\bot_{Y,Z})^\dagger \circ f : x \mapsto f(x) \xrightarrow{\mathcal{P}Y(\bot_{Y,Z})} \{\}, \{\} = \{\}, \{\} \xrightarrow{\mu_{Y^2}} \{\}
\]

As an example of right-strictness, consider $W = \{w\}$; then $\bot_{W,X} : w \mapsto \{\}$, and vacuously, applying $f$ to every element of the empty set $\{\}$ gives $\{}$ again; thus $(f)^\dagger \circ \bot_{W,X} = \bot_{W,Y}$.

---

3This means a set with a partial order structure $\subseteq$, and least upper bounds of $\omega$-chains $x_1 \subseteq x_2 \subseteq \ldots$ which we write $\sqcup_{n<\omega}x_n$. Pointedness means there is a bottom element $\bot$. 

Example 2.2.14. The category $\text{Cpo}_\perp$ is itself $\text{Cpo}_\perp$-enriched; the $\omega$-continuous functions $X \to Y$ may be made into a $\text{Cpo}_\perp$ under a pointwise ordering, where $f \sqsubseteq g$ if $f(x) \sqsubseteq g(x)$ for all $x$ in $X$. The least arrow $\perp_{X,Y} : X \to Y$ maps everything in $X$ to the bottom element of $Y$. Lastly, the join $f$ of an $\omega$-chain $(f_n : X \to Y)_{n<\omega}$ is given by taking joins pointwise: $f(x) = \bigsqcup_{n<\omega} f_n(x)$. In a similar way, one may show that $\text{Cpo}_\perp!$ is $\text{Cpo}_\perp!$-enriched; left- and right-strictness of composition follow from the fact that all arrows preserve the bottom element (whereas they need not in $\text{Cpo}_\perp$).

We will also require that the behaviour functor $\overline{B}$ be compatible with this order structure:

**Definition 2.2.15.** A functor $\overline{B}$ on a $\text{Cpo}_\perp$-enriched category $\mathcal{C}$ is locally monotone if for all arrows $f, g : X \to Y$, $f \sqsubseteq g$ implies $\overline{B}f \sqsubseteq \overline{B}g$. It is locally continuous if for all $\omega$-chains $f_1 \sqsubseteq f_2 \sqsubseteq \ldots$, we have $\overline{B}(\bigsqcup_{n<\omega} f_n) = \bigsqcup_{n<\omega} (\overline{B}f_n)$.

Local continuity implies local monotonicity, via the chain $f \sqsubseteq g \sqsubseteq g \sqsubseteq \ldots$.

Now we outline the main result of [HJS07]: the final $\overline{B}$-coalgebra in $\text{Kl}(M)$ is obtained from the initial $B$-algebra from the underlying category.

**Theorem 2.2.16.** On a category $\mathcal{C}$, let $B$ be a functor which preserves $\omega$-colimits. (As described in Remark 2.1.8, this implies it has an initial algebra $(D, \alpha : B\overline{D} \to \overline{D})$ in $\mathcal{C}$, and the structure $\alpha$ is an isomorphism with inverse $\alpha^{-1} : \overline{D} \to B\overline{D}$).

Moreover, let $M$ be a monad such that $\text{Kl}(M)$ is $\text{Cpo}_\perp$-enriched with left-strict composition, and such that $B$ has a locally monotone lifting $\overline{B}$ to $\text{Kl}(M)$. Then the final $\overline{B}$-coalgebra has carrier $\overline{D}$, and coalgebra-structure $\zeta$ given by $J\alpha^{-1}$. (By definition of $J$, the underlying arrow of $J\alpha^{-1}$ is given by the composition $\overline{D} \xrightarrow{\alpha^{-1}} B\overline{D} \xrightarrow{\eta_{B\overline{D}}} MB\overline{D}$).

As shown, throughout this thesis we will use the symbols $\overline{D}$ and $\overline{\zeta}$ for the carrier and coalgebra structure of the final $\overline{B}$-coalgebra, to distinguish them from the carrier $D$ and structure $\zeta$ of the final $MB$-coalgebra which plays a part in our proofs of adequacy and compositionality at the end of Chapters 3 and 4. In a similar vein, given a $\overline{B}$-coalgebra $(X, \gamma)$, we will write $\overline{\beta}_\gamma$ for the $\overline{B}$-coalgebra morphism $X \to \overline{D}$, and (neglecting the use of decoration $'$) use the same symbol $\overline{\beta}_\gamma$ for the corresponding arrow in the underlying category, of type $X \to MD$. The bar in $\overline{\beta}_\gamma : X \to MD$ distinguishes it from the final $MB$-coalgebra morphism $\beta_\gamma : X \to D$, obtained by construing $(X, \gamma)$ as an $MB$-coalgebra instead of a $\overline{B}$-coalgebra.

The proof of this theorem involves two stages. First, one shows that the initial sequence for $B$ ‘lifts’ via the functor $J$ into the Kleisli category, and maps the initial $B$-algebra $(\overline{D}, \alpha : \overline{D} \to B\overline{D})$ into an initial $B$ algebra $(\overline{D}, J\alpha : \overline{D} \to B\overline{D})$. Secondly, the order structure of $\text{Kl}(M)$ allows this sequence to be reversed, by considering embedding-projection pairs; this yields the final sequence, and the initial $\overline{B}$-algebra as colimit becomes the final $\overline{B}$-coalgebra as limit.
First, one applies the functor $J$ to the initial sequence for $B$, giving the left diagram below, and repeatedly applies the equation $BJ = J\overline{B}$ (on arrows and objects) which asserts that $B$ is a lifting, to obtain the right-hand diagram. As $J$ is a left adjoint, it preserves the initial object, so that $J0 = 0$ is also initial in the Kleisli category, and the initial arrows in the Kleisli category $\text{Kl}(M)$ – which we temporarily call $?_X : 0 \to X$ – are given by applying $J$ to the initial arrows $?_X : 0 \to X$ in $C$. In addition, $J$ preserves the colimit $\overline{D}$ of the initial sequence (and is identity-on-objects), so that $J\overline{D} = \overline{D}$ is the colimit of this diagram, with embeddings $Ji_n : \overline{B^n}0 \to \overline{D}$ as shown. This shows that $\overline{D}$ is also a colimit for the initial $\overline{B}$-sequence.

Using the fact that $B$ preserves $\omega$-colimits, one may show that $\overline{BD}$ is also a colimit of the initial $\overline{B}$-sequence; thus, the isomorphism of colimiting cocones $\alpha : B\overline{D} \cong \overline{D}$ characterising the initial $B$-algebra (Remark 2.1.8) is sent by $J$ to a corresponding isomorphism between the colimiting cocones $J\alpha : B\overline{D} \cong \overline{D}$ (with inverse $J\alpha^{-1}$); this implies that $(\overline{D}, J\alpha : \overline{D} \to B\overline{D})$ is the initial $\overline{B}$-algebra.

Now, to reverse the diagram, one exploits the $\text{Cpo}_{\perp}$-enriched structure of $\text{Kl}(M)$ by introducing the notion of embedding-projection (e-p) pairs $(e : X \to Y, p : Y \to X)$ such that $p \circ e = \text{id}$ and $e \circ p \subseteq \text{id}$. The initial arrow $?^{\prime}_{\overline{B}0} : 0 \to \overline{B}0$ is an embedding with projection $\perp_{\overline{1}0,0} : \overline{B}0 \to 0$. Locally monotone functors $\overline{B}$ preserve e-p pairs, so that $\overline{B}^{?^{\prime}_{\overline{B}0}}$ is also an embedding. Thus, the initial $\overline{B}$-sequence is a series of embeddings $\overline{B}^n_{?^{\prime}_{\overline{B}0}}$, with corresponding projections $\overline{B}^n \perp_{\overline{1}0,0}$.

Such colimits are equivalent to $\text{O}$-colimits [SP82], where the arrows $e_n := Ji_n : \overline{B^n} \to \overline{D}$ into the colimit are also embeddings, and the corresponding projections $p_n$ make $(e_n \circ p_n : \overline{D} \to \overline{D})_{n<\omega}$ an increasing sequence with join $\text{id}_{\overline{D}}$. By considering the corresponding diagram of projections, one obtains a reversed diagram, shown below, and the $\text{O}$-colimit becomes an $\text{O}$-limit, which may be shown to correspond to an ordinary limit of this diagram. The reversed diagram is in fact the final $\overline{B}$-sequence as shown below; the projection $\perp_{\overline{1}0,0,M0} : \overline{B}0 \to 0$ is also the final arrow $\perp_{\overline{B}0} : \overline{B}0 \to 0$ into 0. This implies that $\overline{D}$ is the limit of the final $\overline{B}$-sequence; using the fact that the isomorphism of cocones $J\alpha : B\overline{D} \cong \overline{D}$ is also an e-p pair, a similar argument shows that $J\alpha^{-1} : \overline{D} \cong B\overline{D}$ is also an isomorphism of cones, and implies that $(\overline{D}, J\alpha^{-1})$ is a final $\overline{B}$-coalgebra.
For the monad $M$, a trace prefixes a trace $A$ all empty trace completed traces. The colimit is the collection $\text{here, every occurrence of } 1 \text{ corresponds to a terminal transition; the terms } A^n \times 1 \text{ describe completed traces. The colimit is the collection } \bar{D} = A^* \text{ of finite, completed traces; as } B \text{ is a polynomial functor on } \text{Set} \text{ (which preserves limits of } \omega^{op} \text{ chains), this colimit is the initial } B\text{-algebra, where the isomorphism } \alpha : A \times A^* + 1 \xrightarrow{\sim} A^* \text{ has two components: the first } A \times A^* \rightarrow A^* \text{ prefixes a trace } A^* \text{ with a label } A, \text{ and the second } 1 \rightarrow A^* \text{ treats immediate termination as an empty trace } \epsilon. \text{ The reverse structure } \alpha^{-1} : A^* \rightarrow A \times A^* + 1 \text{ produces the first label } a_1 \text{ of a trace } a_1 \cdots a_n, \text{ and its tail } a_2 \cdots a_n; \text{ the empty trace } \epsilon \text{ is sent to the terminal value.}

For the monad $M = \mathcal{P}$, one may lift $B$ to a functor $\bar{B}$ on $\text{Kl}(M)$, using the distributive law of Example 2.2.9. The action of $\bar{B}$ on an arrow $f : X \rightarrow \mathcal{P}Y$ is as follows: it produces an arrow $\bar{B}f : A \times X + 1 \rightarrow \mathcal{P}(A \times Y + 1)$ which maps a pair $(a, x)$ to the collection of pairs $(a, y)$ for all $y \in f(x) \subseteq Y$, and the terminal transition $\ast \in 1$ to the empty set. The $\text{Cpo}_{op}$-enrichedness and left-strictness of $\text{Kl}(\mathcal{P})$ required for Theorem 2.2.16 follows from Example 2.2.13; $\bar{B}$ is clearly locally monotone (and locally continuous) under the pointwise inclusion ordering: if $f(x) \subseteq g(x)$ for all $x$, then $\bar{B}f(x) \subseteq \bar{B}g(x)$ for all $x$.

As a result, the final $\bar{B}$-coalgebra is given by $(A^*, J\alpha^{-1})$, with underlying structure $\eta_{A^*} \circ \alpha^{-1} : A^* \rightarrow \mathcal{P}(A \times A^* + 1)$. Any non-deterministic lts $(X, \delta : X \rightarrow MBX)$, viewed as an $MB$-coalgebra, may also be considered a $\bar{B}$-coalgebra $(X, \delta' : X \rightarrow \bar{B}X)$. Thus, there is a unique $\bar{B}$-coalgebra morphism $\bar{\beta}_{\delta'} : X \rightarrow \bar{D}$ into the final coalgebra, making the below-left

---

\[
\begin{array}{c}
0 \xleftarrow{\nu_{B0}} B0 \xrightarrow{\nu_{B0}} B2 \xrightarrow{\nu_{B0}} B0 \xrightarrow{\nu_{B0}} \ldots \\
p_0 \quad p_1 \quad p_2 \quad p_3 \quad p_4
\end{array}
\]
diagram commute. This morphism has underlying type $X \rightarrow M\overline{D}$; using the post-composition properties of the arrow $J\alpha^{-1}$ from Remark 2.2.4 (in particular that $(J\alpha^{-1})^\dagger = M\alpha^{-1}$), the underlying diagram in $C$ is below-right.

\[
\begin{array}{ccc}
X & \xrightarrow{\overline{\beta}_{g'}} & \overline{D} \\
\downarrow{\delta'} & & \downarrow{J\alpha^{-1}} \\
\overline{B}X & \xrightarrow{\overline{MB}\overline{\beta}_{g'}} & \overline{B}\overline{D}
\end{array}
\quad
\begin{array}{ccc}
X & \xrightarrow{\overline{\gamma}_{g'}} & M\overline{D} \\
\downarrow{\delta} & & \downarrow{M\alpha^{-1}} \\
MBX & \xrightarrow{MB\overline{\beta}_{g'}} & MB\overline{D} & \xrightarrow{M\lambda_{\overline{D}}} & M^2B\overline{D} & \xrightarrow{\mu_{B\overline{D}}^{\dagger}} & MB\overline{D}
\end{array}
\]

In our running example (where $M = \mathcal{P}$, $BX = A \times X + 1$, and $\overline{D} = A^*$), one might expect that the unique coalgebra morphism $\overline{\beta}_{g'} : X \rightarrow \overline{D}$ will map each element $x \in X$ to the collection of traces $\{t_i : i \in I\} \subseteq A^*$ it exhibits. To verify this guess, it is enough to check directly that this choice of $\overline{\beta}_{g'}$ makes the diagram commute; by finality, there is a unique map with this property.

With this definition of $\overline{\beta}_{g'}$, the interpretation of the right-hand diagram is as follows. Post-composing $\overline{\beta}_{g'}$ with $P\alpha^{-1}$ has the effect of observing the first transition of each trace $t_i$ exhibited by $x$, producing either: a pair $(a, t'_i)$ of the first label $a$ and the tail $t'_i$; or a terminal transition $\ast \in 1$, if the trace is empty.

As for the bottom-left path, the left-hand arrow $\delta$ unfolds the transition behaviour of the state $x \in X$, producing a set of pairs $(a, x')$ and/or a terminal transition *. Again, each successor state $x'$ is mapped by $MB\overline{\beta}_{g'}$ to the set of traces $t'_i$ it exhibits; and the maps $\lambda_{B\overline{D}}$ and $\mu_{\overline{D}}$ take the union of these sets of traces.

The diagram then amounts to the assertion that the $a$-prefixed traces $at'_i \in \overline{\beta}_{g'}(x)$ of the state $x$ are in 1-1 correspondence with the traces $t'_i$ of its $a$-transition successors; and that if $x$ may terminate, then its set of traces $\overline{\beta}_{g'}(x)$ contains the empty trace $\epsilon$. As this is the case, the diagram commutes for our chosen definition of $\overline{\beta}_{g'}$, which must hence be the final $\overline{B}$-coalgebra morphism from $X$ into $\overline{D}$.

**Final Kleisli Coalgebras By Algebraic Compactness**

For our purposes, one disadvantage of the above limit-colimit result is that it does not explicitly characterise the coalgebra morphisms $\overline{\beta}$ into the final coalgebra; one must manually check that the expected behavioural equivalence makes the above diagrams commute. An alternative approach is to obtain the final-coalgebra morphisms by a least-fixpoint construction. The required result, proved on p.98 of [Joh92], is as follows:

**Proposition 2.2.18.** Let $\mathcal{D}$ be a $\mathbb{Cp}_\perp$-enriched category with left-strict composition, and $G$ a locally-continuous endofunctor on $\mathcal{D}$. An initial $G$-algebra $(\overline{D}, \theta : G\overline{D} \cong \overline{D})$, if it exists, yields
a final G-coalgebra \((\overline{D}, \theta^{-1} : \overline{D} \cong GD)\). Given a G-coalgebra \(\gamma : X \to GX\), the corresponding unique G-coalgebra morphism \(\overline{\beta}_\gamma : X \to \overline{D}\) is the least fixpoint of this operator (on arrows):

\[
\Phi : (X \xrightarrow{f} \overline{D}) \mapsto (X \xrightarrow{\gamma} GX \xrightarrow{Gf} GD \xrightarrow{\theta} \overline{D}).
\]

Thus (by the Knaster-Tarski theorem), \(\overline{\beta}_\gamma\) is the join of the approximants \(\overline{\beta}_\gamma^{(n)} := \Phi^n(\bot_X, \overline{D})\) for \(n < \omega\).

To apply this result in our setting, we take \(D = \text{Kl}(M)\) and \(G = B\), and make the assumptions of Theorem 2.2.16, in addition to assuming that \(B\) is locally continuous. These assumptions guarantee existence of the initial \(B\)-algebra \((\overline{D}, \theta)\), where \(\theta = J\alpha\). The previous result then implies the final \(B\)-coalgebra has structure \(\theta^{-1} = J\alpha^{-1}\); moreover, the approximants \(\overline{\beta}_\gamma^{(n)}\) give a concrete characterisation of the behavioural equivalence induced by mapping into the final coalgebra.

**Example 2.2.19.** We illustrate the trace semantics induced for our running example where \(M = \mathcal{P}, BX = A \times X + 1\), and \(\overline{D} = A^*\). Given an MB-coalgebra \((X, \gamma)\), the underlying arrow of the final coalgebra morphism \(\overline{\beta}_\gamma : X \to \mathcal{P}(A^*)\) is the join of the approximants \(\overline{\beta}_\gamma^{(n)}\) described below – meaning simply that \(\overline{\beta}_\gamma(x)\) is the union of the sets \(\overline{\beta}_\gamma^{(n)}(x)\) for all \(n < \omega\).

The first approximant \(\overline{\beta}_\gamma^{(0)} : X \to \mathcal{P}(A^*)\) simply assigns every state \(x\) an empty set. Then successive approximants act as follows: given approximant \(\overline{\beta}_\gamma^{(n)}\), the next approximant has underlying arrow \(\overline{\beta}_\gamma^{(n+1)} = (J\alpha^{-1})^\dagger \circ (B\overline{\beta}_\gamma^{(n)})^\dagger \circ \gamma\), which we break down as follows:

\[
\overline{\beta}_\gamma^{(n+1)} : X \xrightarrow{\gamma} \mathcal{P}(A \times X + 1) \xrightarrow{\mathcal{P}B\overline{\beta}_\gamma^{(n)}} \mathcal{P}(A \times \mathcal{P}(A^*) + 1) \xrightarrow{\mu_{BA^*} \circ \mathcal{P}B\lambda_{A^*}} \mathcal{P}(A \times A^* + 1) \xrightarrow{(J\alpha^{-1})^\dagger = \mathcal{P}a^{-1}} \mathcal{P}(A^*)
\]

We now show that \(\overline{\beta}_\gamma^{(1)}\) records if \(x\) has any ‘traces of length 0’: it gives us a set containing the empty trace \(\{\epsilon\}\) if \(x\) has a terminal transition, and empty otherwise; any other traces are ignored. In the above composition, the second approximant \(\overline{\beta}_\gamma^{(1)}\) begins by unfolding the transition behaviour of each state \(x \in X\), giving a set of \(a\)-transitions represented by pairs \((a, x')\), and/or the terminal value \(* \in 1\) (if \(x\) may terminate). Each successor state \(x'\) in this set is sent by the previous approximant \(\overline{\beta}_\gamma^{(0)}\) (in the map \(\mathcal{P}B\overline{\beta}_\gamma^{(0)}\)) to the empty set, and terminal values \(*\) are unaffected. The maps \(\mu_{BA^*} \circ \mathcal{P}B\lambda_{A^*}\) remove all the empty collections \((a, \{\})\); the resulting element of \(\mathcal{P}(A \times A^* + 1)\) is then either an empty set, or the singleton \(\{*\}\) if \(x\) had a terminal transition. (The map \(\mathcal{P}a^{-1}\) then ‘wraps’ any occurrence of the terminal value \(*\) into an empty trace \(\epsilon\).)

We now consider the third approximant \(\overline{\beta}_\gamma^{(2)}\). Similarly, it unfolds the transition behaviour of \(x\) into a collection of pairs \((a, x)\) and/or \(* \in 1\). This time, the previous approximant \(\overline{\beta}_\gamma^{(1)}\) gives
more information about the successor states $x'$; it produces a set $\{\epsilon\}$ if they may terminate, and the empty set $\{}$ otherwise. The map $(B\beta^{(n)})^!$ (given by the second and third arrows above) now produce a collection of pairs $(a, \epsilon)$ corresponding to $x$'s completed traces of length 1, in addition to any terminal transitions $\ast$ (traces of length 0) $x$ might have had. Finally, the inverted algebra-structure $\alpha^{-1}$ in $\mathcal{P}\alpha^{-1}$ maps the pairs $(a, \epsilon)$ into traces $a$ of length 1, and wraps the terminal value $\ast$ into the empty trace $\epsilon$ of length 0. Thus, $\beta^{(2)}$ produces a subset of $\mathcal{P}(A^*)$ containing the traces of length 1 or less.

In a similar manner, successive approximants $\beta^{(n)}$ map state $x$ to its completed traces of length $n - 1$; and the union of these sets, i.e. $\beta(x)$, is the collection of all finite traces of the state $x$.

### 2.3 Effects, State, and Comodels

We have considered non-deterministic behaviour in the context of process algebras. This may be seen as an instance of a more general phenomenon of *effects* in semantics, whereby one separates program execution from its interaction with external phenomena, such as variables in a store, user input, or non-deterministic choice. We review some results from this research programme.

The idea of side-effects was famously formalised by Moggi in his ‘notions of computation’ paper [Mog91], where a monad describes the effects present in a particular language. However, monads give a rather coarse-grained semantic description of effects; for instance, it is difficult to explain how monads combine (via monad transformers) to describe the situation when several kinds of effects are present. This led to a research programme formulating such monads in the more foundational terms of *computational effects*, characterised by Lawvere theories.

#### 2.3.1 Monads and Notions of Computation

The idea proposed by Moggi was to characterise a computation $f' : X \to Y$, taking inputs of type $X$ and producing outputs of type $Y$, as a function of type $f : X \to MY$ for some monad $M$ – thus, an arrow in the Kleisli category $\text{Kl}(M)$. Composition in the Kleisli category gives a natural way of chaining together computations $X \xrightarrow{f'} Y \xrightarrow{g'} Z$; the idea is that the effects from each computation are accumulated, and combined by the multiplication $\mu_Z$ of the monad $M$.

We will use three running examples of effects:

1. **Global Store.** In languages with global store, computations $f' : X \to Y$ are equivalent to functions $f : (X \times S) \to (Y \times S)$, for some notion of store $S$. Such a computation takes a pair $\langle x, s \rangle$ of an initial value $x$ and store $s$, and returns a pair $\langle y, s' \rangle$ of new value $y$ and store $s'$. 
Thus, different computation paths may occur depending on the store \( s \), which in turn may be updated during program execution.

By currying, we may re-express such a computation as a function \( f_c : X \to (Y \times S)^S \), and hence an arrow \( f_c : X \to MY \), where \( M \) is the side-effect monad \( MX = (S \times X)^S \). Its unit \( \eta_X : X \to MX \) takes a value \( x \), and produces the computation that ‘returns \( x \) and ignores the store’: \( \eta_X(x) = \lambda s.(x,s) \). Multiplication \( \mu_X \) substitutes the output store from the outer occurrence of \( M \) into the input of the inner occurrence.

\[
\mu_X : ((X \times S)^S \times S)^S \to (X \times S)^S \quad g \mapsto \lambda s. \text{let } \langle f : (X \times S)^S, s : S \rangle \leftarrow g(s) \text{ in } f(s)
\]

A common choice of store is given by \( S = \mathbb{N}^L \): assignments of natural number values to variables, indexed by a given set of locations \( L \). Example computations include returning the value of a variable \( x \in L \): \( \langle \text{return } x, s \rangle \mapsto \langle (s(x), s) \rangle \), of computation type \( L \to \mathbb{N} \). Another would be to update the store at a given location \( L \) with a given value in \( \mathbb{N} \): \( \langle x = n, s \rangle \mapsto \langle \ast, s[x \mapsto n] \rangle \) (where \( \ast \in 1 \) represents a void return value, and \( s[x \mapsto n] \) is the store \( s \) updated so that \( x \) has value \( n \)). This is a computation of type \( L \times \mathbb{N} \to 1 \).

One composes two functions in the natural manner, corresponding to simple composition of the uncurried functions \( (X \times S) \xrightarrow{f} (Y \times S) \xrightarrow{g} (Z \times S) \); the output value and store of the first program are used as the input of the second, yielding a final value-and-store pair; and one may check that this is the effect of composing the curried arrows in the Kleisli category.

2. Non-determinism. We have already considered non-determinism, where \( M \) is the power-set monad \( P \). A computation \( f' : X \to Y \), given by an arrow \( f : X \to PY \), assigns to each input \( x \in X \) a set of outputs \( f(x) = s \subseteq Y \). As illustrated in Example 2.2.2, to compose computations \( g' \circ f' \), one evaluates the underlying function \( g \) of the second computation \( g' \) on each output \( y \) in \( f(x) \), giving a set of subsets of \( Z \); the multiplication \( \mu_Z \) takes their union, giving every possible result of executing \( f \) followed by \( g \).

3. Non-determinism with Global Store. Programs exhibiting both kinds of effects may be described by the monad \( MX = (P(X \times S))^S \). A computation \( f' : X \to Y \) is a function \( f : (X \times S) \to P(Y \times S) \), assigning to each initial value-store pair \( (x,s) \) the collection of possible new value-store pairs \( (y,s') \). As before, currying gives a function \( f_c : X \to MY \). Computations \( X \xrightarrow{f'} Y \xrightarrow{g'} Z \) are composed in the natural way: uncurrying the functions \( f, g \), and applying \( f \) to an initial value and store \( (x, s) \), yields a set of pairs \( (y, s') \) in \( (Y \times S) \). Then each pair is used as the input of \( g \), producing a further collection of pairs \( (z, s'') \) in \( (Z \times S) \); finally, the multiplication of the power-set monad takes the union of all the sets \( g(y, s') \) for all \( (y, s') \in f(x, s) \).

Finally, although we will not consider the following side-effects in detail, we will refer to them...
4. Exceptions. A program may evaluate incorrectly, producing some exception $e$ from a set $E$. The resulting exceptions monad is $MX = X + E$, with unit $\eta_X = \text{inl}_X$ and multiplication $\mu_X = [\text{id}, \text{inr}_E] : (X + E) + E \rightarrow X + E$. A computation $X \rightarrow Y + E$ returns either a value $y \in Y$, or an exception $e \in E$; and composing programs $f', g'$ simply produces an exception $e$ if either $f(x)$ or $g(f(x))$ does, otherwise it produces the value of $g(f(x))$.

5. Interactive Input/Output (I/O). This refers to two atomic possibilities: a computation $X \rightarrow Y$ may output a ‘character’ or label $o$ from some set $O$ (in addition to some $y \in Y$); or request that the user input a label $i$ from a (finite) set $I$, before returning a $Y$. Such computations correspond to functions $X \rightarrow \Theta Y$, where $\Theta Y = O \times Y + Y^I$; however, for successive computations to accumulate I/O effects, one must permit a sequence of such actions; this corresponds to taking the monad $M$ to be free $\Theta$-algebra functor $T_\Theta$ (like the free $\Sigma$-algebra functor $T_\Sigma$ for constructing program syntax, of Section 2.1.3). A general I/O computation is then an arrow $X \rightarrow T_\Theta Y$, exhibiting sequences of I/O actions, and composition of such arrows corresponds to sequencing the I/O actions.

2.3.2 Computational Effects

A more foundational approach to effects involves characterising the computational monads in terms of algebraic theories. Given a polynomial syntax functor $\Sigma$, we have already shown how a syntactic monad $T = T_\Sigma$ constructs freely generated $\Sigma$-algebras $TX$ over variables $X$, corresponding to syntax terms over $X$. Indeed, several computational monads arise in this way.

User I/O. We have already defined the I/O monad as the free $\Theta$-algebra functor $T_\Theta$, for output labels $O$ and input labels $I$. Such algebras consist of unary operators $\text{out}_o(v)$ for every $o \in O$, describing an output of the symbol $o$ before producing value $v$; and an $I$-ary operator $\text{in}((v_i)_{i \in I})$ which asks the user for an input, and produces value $v_i$ if the user enters input $i$.

Exceptions. The exceptions monad $MX = X + E$ is simply given by adding nullary effects $e \in E$ to $X$, so that $M = T_\Sigma$ where $\Sigma = E$ is a constant functor.

However, many computational monads, such as those for non-determinism and global store, do not correspond to freely-generated algebras; an algebraic theory is required, involving effect operations and semantic equations.

(Finitary) Non-determinism. In $\text{Set}$, the finitary power-set monad $\mathcal{P}_1$ can be described by the algebraic theory of semilattices, containing a single (infix) binary operation $v_1 \triangledown v_2$, to be read as a non-deterministic branch, giving either value $v_1$ or $v_2$. The theory has three equations: $x \triangledown x = x$ (idempotence); $x \triangledown y = y$ or $x$ (symmetry); and $x \triangledown (y \triangledown z) = (x \triangledown y) \triangledown z$. 
(associativity). Given a collection of variables \( X \), effect syntax over \( X \) is given by binary \( \text{or} \)-trees with leaves given by variables \( x \in X \); each tree represents a collection of possible values of \( X \). For instance, both syntax-trees \(( y \text{ or } x)\) and \(( x \text{ or } (x \text{ or } y))\) correspond to the set of values \( \{x, y\}\). The equations ensure that the exact shape of each tree is irrelevant; they equate two trees if and only if their leaves correspond to the same subset of \( X \). The terms in the free algebra over \( X \) thus correspond to finite subsets of \( X \), i.e. elements of \( \mathcal{P}f(X) \). To lift the finitary restriction, one may extend the algebraic theory with a ‘wide \( \text{or} \)’ of countable arity, with some additional equations; but in Chapter 4, we will find that in an ordered setting, the models are given by convex powerdomains, which will also include countable sets.

**Global Store.** A milestone in the research on computational effects was the identification of an algebraic theory for the monad for global store, where \( S = \mathbb{N}^L \) [PP02]. This requires two kinds of operation: the first is a unary ‘write’ operation \( \text{wr}_{x,n}(v) \) which represents an update to the store, by setting location \( x \) to value \( n \), before producing the value \( v \). The second kind of effect is a \( \mathbb{N} \)-ary ‘read’ operation \( \text{rd}_x(v_1, v_2, v_3, \ldots) \), which we also write as \( \text{rd}_x((v_n)_{n\in\mathbb{N}}) \). Analogously to the I/O input operation, it represents a branch in the computation depending on the value of the variable \( x \) in the store, returning the value \( v_n \) if \( x = n \).

For instance, an expression returning the value of \( x \) in the store would be represented by the read operation \( \text{rd}_x(0, 1, 2, \ldots) = \text{rd}_x((n)_{n\in\mathbb{N}}) \). An update \( x = 5 \) would be represented by \( \text{wr}_{x,5}(*) \) (where, as before, \(* \) is a void return value). Combining both effects, the update \( x = y \) would correspond to the tree \( \text{rd}_y((\text{wr}_{x,n}(*))_{n\in\mathbb{N}}) \).

There are seven equations ensuring the correct semantics of reads and writes. Two examples are: \( \text{wr}_{x,m}(\text{wr}_{y,n}(v)) = \text{wr}_{y,n}(\text{wr}_{x,m}(v)) \) for all \( x, y \in L \) with \( x \neq y \), expressing that updates of different variables \( x, y \) are order-interchangeable; and \( \text{wr}_{x,n}(\text{wr}_{x,m}(v)) = \text{wr}_{x,m}(v) \), expressing that in a series of writes to a variable \( x \), it is only the final update that counts. It is shown in [PP02] how the algebras of this equational theory give rise to the side-effect monad for global store, expressed in more categorical generality via an \( S \)-fold coproduct \( MX = (S \cdot X)^S \) rather than a product in \( \text{Set} \), viz. \( MX = (S \times X)^S \). We will instantiate the more general definition in the categorical setting of \( \text{Cpo}_{\perp!} \) in Chapter 4.

**Algebraic Effects via Lawvere Theories**

The categorical machinery used to study algebraic theories of effects is given by *countable Lawvere theories* [Pow06]. In an unenriched setting, these are categories whose objects \( n \) are essentially sets \( \{1, \ldots, n\} \), each of which formally represents an ordered \( n \)-tuple of arguments. An effect \( e \) with arity \( n \) is represented by a formal arrow \( e : n \to 1 \). For instance, the binary \(4\)More generally, one may consider a family of \( m \) such effects as a single arrow \( e : n \to m \), but out of convenience we will only refer to effect-arrows of type \( n \to 1 \) for some \( n \).
or operator of non-determinism is represented by an arrow \( \textbf{or} : 2 \rightarrow 1 \) in a Lawvere theory.

Such Lawvere theories are built upon a category \( \aleph_1^{op} \) giving appropriate structure to these tuples – where \( \aleph_1 \) is a skeleton of the category \( \text{Set}_f \) of countable sets [HPP06], of which we give an explicit example.

**Definition 2.3.1.** Let \( \aleph_1 \) be the category which has an object \( n \) for every ordinal \( 0 \leq n \leq \omega \) given by a set \( \{1, \ldots, n\} \) (including \( 0 = \{\} \) and \( \omega = \{1,2,\ldots\} \)), and no others; and whose arrows \( n \rightarrow m \) are given by all functions between these sets.

The intention is for each arrow \( f : m \rightarrow n \) of a Lawvere theory \( \mathcal{L} \) to describe ‘a way of producing an \( n \)-tuple of arguments from an \( m \)-tuple’ – for instance, it might rearrange, discard, or repeat arguments of the \( m \)-tuple. This means that for every \( r \in \{1, \ldots, n\} \), we must choose which of the original arguments \( \{1, \ldots, m\} \) it came from. Thus, \( f \) corresponds to a set-function in the opposite direction: an arrow \( n \rightarrow m \) in \( \aleph_1 \). This is why we must reverse the arrows: we essentially build Lawvere theories on top of \( \aleph_1^{op} \).

This kind of tuple-manipulation may be described more abstractly in terms of products, as follows. In \( \aleph_1 \), the coproduct of \( m \) and \( n \) is \( m + n \); in \( \aleph_1^{op} \), it becomes a product. In particular, \( m \) becomes the \( m \)-fold product of \( 1 \), with \( r \)th projection \( p_r : m \rightarrow 1 \) which essentially discards all arguments except the \( r \)th (so that \( p_r \) is given by the corresponding \( \aleph_1 \)-arrow \( 1 \rightarrow m \) mapping \( 1 \) to \( r \)). By taking \( n \)-fold products of these projections, one obtains arrows \( m \rightarrow n \) which formally describe all ways of obtaining \( n \) arguments from \( m \) arguments, as intended.

To this category, a Lawvere theory \( \mathcal{L} \) essentially adds arrows \( e : n \rightarrow 1 \) formally representing effects, along with all the derived operations given by closure under composition. To add arrows to \( \aleph_1^{op} \), one asks for an identity-on-objects functor \( \aleph_1^{op} \rightarrow \mathcal{L} \); to ensure the tuples may still be manipulated in terms of products in \( \mathcal{L} \), one asks for strict product preservation\(^5\), leading to the following definition.

**Definition 2.3.2.** A countable Lawvere theory is a category \( \mathcal{L} \) with countable products and a countable strict-product-preserving, identity-on-objects functor \( F : \aleph_1^{op} \rightarrow \mathcal{L} \). (Note that due to the presence of the ordinal \( \omega \), \( \aleph_1^{op} \) has countable products.)

Equations on the effects are specified by sketches [BW85], which correspond to commuting diagrams in the Lawvere theory \( \mathcal{L} \). As an example, the idempotence equation \( x \textbf{or} x = x \) for nondeterminism is represented by commutativity of the left-below diagram, where the pair \( \langle \text{id}_1, \text{id}_1 \rangle : 1 \rightarrow 2 \) duplicates an argument, before passing the pair into \( \textbf{or} \). The right-hand

\(^5\)Regarding strictness of product preservation for Lawvere theories (and non-strictness for models, in Definition 2.3.3), see [HP07] pp.5-6.
diagram is the sketch for commutativity, which asserts that \texttt{swap}-ping the arguments of \texttt{or} has no effect.

\[
\begin{align*}
1 & \xrightarrow{\text{id}_1} 2 \\
\text{id}_1 & \downarrow \text{or} \\
1 & \quad 1 \\
\text{or} & \quad \text{or} \\
\end{align*}
\]

(2.2)

For the theory of global store, the seven equations and corresponding diagrams are in [PP02].

\textbf{Models of Lawvere Theories}

In Sections 2.1.1, we considered polynomial syntax-functors \( \Sigma \) given by collections of operations, and showed how a \( \Sigma \)-algebra \((A, \gamma)\) corresponds to a carrier \( A \) with functions \( \gamma_n : A^n \to A \) for each \( n \)-ary operation. Generalising this idea, we may describe models of a Lawvere theory \( \mathcal{L} \) in a category \( \mathcal{C} \), as product-preserving functors.

\textbf{Definition 2.3.3.} If \( \mathcal{C} \) has countable products, the \textit{category of models} \( \text{Mod}(\mathcal{L}, \mathcal{C}) \) of \( \mathcal{L} \) in \( \mathcal{C} \) has as objects all countable product-preserving functors \( P : \mathcal{L} \to \mathcal{C} \), and as arrows all natural transformations between them.

One may describe the construction of \textit{free models} of a Lawvere theory in terms of an adjunction. The forgetful functor \( U^\mathcal{L} : \text{Mod}(\mathcal{L}, \mathcal{C}) \to \mathcal{C} \) maps a model \( P \) to its carrier, \( P1 \), and a natural transformation between models \( \epsilon : P \to Q \) is mapped to its 1-component \( \epsilon_1 : P1 \to Q1 \). The functor \( U^\mathcal{L} \) has a left-adjoint \( F^\mathcal{L} : \mathcal{C} \to \text{Mod}(\mathcal{L}, \mathcal{C}) \) if the category \( \mathcal{C} \) is locally countably presentable [AR94]; \( F^\mathcal{L}X \) may be thought of as the free model of the Lawvere theory \( \mathcal{L} \) with generators \( X \). The monad induced by the effect theory is then given by the composition \( M = U^\mathcal{L} F^\mathcal{L} \). In this way, one recovers the notions of computation in Section 2.3.1: the finite power-set monad \( P_f \) from the equational theory for non-determinism, and the side-effect monad from the theory for global store.

\textbf{Combinations of Lawvere Theories}

One advantage of formulating effects in terms of Lawvere theories, rather than monads, is a more satisfactory treatment of modularity. Given two theories corresponding to different kinds of effects, one may combine them via a \textit{commutative sum} \( \mathcal{L}_1 + \mathcal{L}_2 \), which combines the operations and equations of both; another combination is the \textit{tensor} \( \mathcal{L}_1 \otimes \mathcal{L}_2 \), which requires in addition that every operation of either theory commutes with the operators of the other [HPP06]. We consider two examples:

\textbf{Global Store and Non-determinism.} This combination of effects is described by the tensor of the theories of non-determinism and global state. It has read and write operations \( \text{rd}_x, \text{wr}_{x,n} \),
indexed by $L$ and $L \times N$ respectively, in addition to the binary choice operator $\text{or}$. There are two commutativity equations (for every location $x \in L$ and natural number $n \in \mathbb{N}$, and for any $\mathbb{N}$-indexed families of syntax variables $(p_m)_{m \in \mathbb{N}}$ and $(q_m)_{m \in \mathbb{N}}$):

\[
\text{wr}_{x,n}(p \text{ or } q) = \text{wr}_{x,n}(p) \text{ or } \text{wr}_{x,n}(q) \quad \text{rd}_x((p_n \text{ or } q_n)_{n \in \mathbb{N}}) = \text{rd}_x((p_n)_{n \in \mathbb{N}}) \text{ or } \text{rd}_x((q_n)_{n \in \mathbb{N}})
\]

The first equation states that it makes no difference whether the store-variable $x$ is updated before or after one makes a choice of either $p$ or $q$; the second makes a similar statement about read operations. These equations guarantee a sensible interaction of non-determinism with lookups and variable updates.

**Global Store and Exceptions.** On the other hand, one employs the commutative sum for combining exceptions with global state (or user I/O). This contains reads, writes, and nullary operations (i.e. constants) $g$ corresponding to exceptions. For instance, a computation that updates $x$ to $n$ before halting with an exception $g$ would produce the effect term $\text{wr}_{x,n}(g)$. The tensor of the theories would additionally stipulate that ‘halting (with an exception) after a variable update is equivalent to immediately halting’: $\text{wr}_{x,n}(g) = g$. This is usually not the desired semantics, hence the sum is used.

The tensor of two theories has an elegant characterisation in terms of categories of models; we will use it in Section 4.1.5 to show how some of the effects $\mathcal{L}_1$ of a tensor $\mathcal{L}_1 \otimes \mathcal{L}_2$ may be passed to an implementation given by a comodel (defined in the next section).

**Theorem 2.3.4.** For any category $\mathcal{C}$ with countable products, there is a coherent equivalence between the categories $\mathcal{L} = \text{Mod}(\mathcal{L}_1 \otimes \mathcal{L}_2, \mathcal{C})$ and $\mathcal{R} = \text{Mod}(\mathcal{L}_1, \text{Mod}(\mathcal{L}_2, \mathcal{C}))$ – meaning a pair of functors $P: \mathcal{L} \to \mathcal{R}$ and $Q: \mathcal{R} \to \mathcal{L}$ defining an equivalence of categories, with the following additional property. Let $U_\mathcal{L}: \mathcal{L} \to \mathcal{C}$ and $U_\mathcal{R}: \mathcal{R} \to \mathcal{C}$ be the respective forgetful functors from these categories of models, taking models to their carriers; then we have $P \circ U_\mathcal{R} = U_\mathcal{L}$ and $Q \circ U_\mathcal{L} = U_\mathcal{R}$.

By using properties of adjunctions, this result may be used to characterise the monads arising from tensors of theories. The combination of global store with other theories is described by the following result:

**Theorem 2.3.5.** Let $S = \mathbb{N}^L$, and let $\mathcal{C}$ be a category with countable (and hence, $S$-fold) powers and copowers. Given a Lawvere theory $\mathcal{L}$, the monad on $\mathcal{C}$ induced by the tensor of $\mathcal{L}$ with the theory for global store is isomorphic to the monad $MX = (N(S \cdot X))^S$, where $N$ is the monad induced by the theory $\mathcal{L}$.

In the setting of $\text{Set}$, the monad $MX = (\mathcal{P}_f(S \cdot X))^S$ obtained by combining global store with non-determinism (whose induced monad is the finitary power-set monad $\mathcal{P}_f$) is isomorphic to the computational monad $MX = (\mathcal{P}_f(X \times S))^S$, as described in Section 2.3.1.
2.3. Effects, State, and Comodels

2.3.3 State via Comodels

Algebraic effects have been used to structure the denotational semantics of functional programming languages. However, the role of effects in the operational semantics of these languages is less well-understood. Operational semantics had largely been studied in terms of syntactic effects [PP01] and [JSV10]; in such treatments, there is no interpretation of how effects are implemented – for instance, what happens when a variable is updated, or when a user interacts with a program. This problem was tackled in [PP08], which identified the importance of formalising the notion of persistent state, such as a store, which evolves as programs execute.

In terms of algebraic effects, it may be described by a comodel of the Lawvere theory; this is essentially a transition system which undergoes transitions as it ‘consumes’ effects. We give two examples.

User I/O. In the context of interactive programs, one could model the internal state of the user as a giant transition system, with some carrier $S$. Whenever an output effect $\text{out}_o(v)$ occurs, this causes the user’s state to change; if the user is asked to input a label $i \in I$ for an input effect $\text{in}((v_i)_{i \in I})$, they must produce a label $i$ – and their state may change in the process.

This means that in every user-state $s$, the user may be made to undergo transitions to new states $s'_o$ (if the user is shown an output $o \in O$), and also $I$-labelled transitions $s \xrightarrow{i} s'$ (if the user is asked to give an input from $I$). Thus, the transitions of the user are fully specified by functions $\text{see}_o : S \rightarrow S$ for every output $o \in O$, and a function $\text{give} : S \rightarrow I \times S$ (for giving an input $i \in I$). By taking products, we may combine these into a single function $S \rightarrow S^O \times (I \times S)$. Defining the functor $\Theta^\alpha X := X^O \times (I \times X)$, we may represent a user of the system as a $\Theta^\alpha$-coalgebra.

Global Store. In a similar manner, an implementation of global store (with state-space $S$) is a transition system which responds to read- and write-effects. In any given state $s$, for every possible update (of location $x \in L$ with value $n \in \mathbb{N}$), there is a corresponding transition to a new store $s \rightarrow s'_{x,n}$. For every location $x$, a lookup on $x$ produces a value $n \in \mathbb{N}$. For convenience, let us say that it also produces a ‘new’ state $s'$ (which should be the same as $s$, in sensible implementations of the store.)

This data implies that every state $s$ has possible transitions $(s'_{x,n})$ for all $x \in L, n \in \mathbb{N}$, which are responses to update requests $x \mapsto n$. These transitions may be described by functions $\text{upd}_{x,n} : S \rightarrow S$, assigning each state $s$ the result $s'$ of requesting that update. In addition, a lookup on a location $x$ corresponds to an $\mathbb{N}$-labelled transition to a ‘new’ state $(t'_x)_{x \in L}$ – represented by a function $\text{lku}_x : S \rightarrow \mathbb{N} \times S$. Thus, the store may be represented as a $G$-coalgebra, where $GX = X^{L \times \mathbb{N}} \times (\mathbb{N} \times X)^L$.

However, in contrast to user I/O, not every such coalgebra is a correct implementation of a
store. The standard axiomatisation of such implementations is given by the theory of arrays, involving two operations closely related to upd and lku, and four equations on them [PS04] which bear a close correspondence with four of the equations for the theory of global store.

We now define comodels, and show how they correspond with the coalgebraic transition systems described above.

**Definition 2.3.6.** Given a countable Lawvere theory $L$, the category of comodels $\text{Comod}(L, C)$ (of $L$ in $C$) has as objects the countable coproduct-preserving functors $C : L^{\text{op}} \to C$, and arrows given by all natural transformations between these functors.

An effect arrow $e : n \to 1$ in the Lawvere theory $L$ corresponds to an arrow $e^{\text{op}} : 1 \to n$ in category $L^{\text{op}}$. Intuitively, whenever an instance of the effect $e((v_m)_{m \in n})$ is encountered, a state-based implementation of the effects must choose a branch $v_m$ – or equivalently, an index $m$ from the set $n$ – and possibly change state in the process. (For unary effects $e^{\text{op}} : 1 \to 1$, no choice is involved; it consumes the effect, and may change state.) In Set, this amounts to a transition function $\gamma : S \to n \times S$ assigning each state $s$ a pair $(m, s')$ of an index $m$ and a new state $s'$. In a more general setting, one replaces the product $n \times S$ with an $n$-fold coproduct $n \cdot S$ of $S$, giving $\gamma : S \to n \cdot S$. An implementation must then provide such an arrow for every $n$-ary effect $e$.

Examples of unary effects are the ‘update’ functions $\text{upd}_{x,n} : S \to S$ implementing writes $\text{wr}_{x,n}$ to global store, and the ‘see’ functions $\text{see}_o : S \to S$ implementing output $\text{out}_o$ in user I/O. By contrast, the $N$-ary lookup implementation $\text{lku}$ may be expressed as a function $S \to N \cdot S$, and similarly, providing user input $\text{give}$ corresponds to a function $S \to m \cdot S$ where $m$ is the cardinality of $I$ (assumed countable!).

We show how a comodel $C$ of $L$ provides such an implementation, where (as for models) the state-space is given by $C1$. In $L^{\text{op}}$, $n$ is the $n$-fold coproduct of $1$ (as in $\aleph_1$): $n = n \cdot 1$. As before, coproduct preservation implies that there is a coherent isomorphism $\theta_n : Cn \cong n \cdot C1$. For every reversed effect arrow $e^{\text{op}} : 1 \to n$, we may take its image $C(e^{\text{op}}) : C1 \to Cn$ under $C$, and post-compose with the isomorphism $\theta_n$, giving the required arrow $C1 \to n \cdot C1$ implementing the effect $e$. We may then take the product of these arrows to turn the comodel into a $\Delta^{\text{co}}$-coalgebra for a suitable functor $\Delta^{\text{co}}$, with carrier $C1$.

### 2.3.4 Constraints on Comodels

When one reverses the arrows of a theory $L$ to give $L^{\text{op}}$, the sketches defining the Lawvere theory $L$ translate into constraints in $L^{\text{op}}$ on the comodels, or co-equations. This reversal essentially replaces effects $e : n \to 1$ with their implementations $e^{\text{op}} : 1 \to n$, and product structure by
coproducts. For instance, some of the equations for global store translate directly into axioms on arrays; it is shown in [PS04] that there is a 1-1 correspondence between arrays and comodels for global store in \textbf{Set}.

However, for theories such as non-determinism, reversing the sketches produces unreasonable constraints, which usually means there are no (non-trivial) comodels. This demonstrates that a theory in terms of comodels alone is insufficient to describe the operational semantics of languages exhibiting such effects.

This also happens for comodels of the theory of exceptions in \textbf{Set}, due to the presence of nullary effects \(e : 0 \to 1\). In models \(P\), these effects correspond to arrows \(P0 \cong 1 \to P1\) which ‘pick out’ the exception constants. The 0-fold coproduct of an object is the initial object 0; so by coproduct preservation, any comodel \(C\) must map the reversed arrows \(e^{op} : 1 \to 0\) into arrows \(C1 \to C0 \cong 0\). In \textbf{Set}, there are no such arrows into the initial object 0, given by the empty set, unless the carrier \(C1\) is also empty. In \(\text{Cpo}\_\bot\) and \(\text{Cpo}_\bot\!,\) the situation is different; the initial object is given by the one-element \(\text{cppo}\_\bot\), and every constant must therefore be interpreted as undefined \(\bot\).

Problems also arise in the presence of non-determinism, where the effect \(\text{or} : 2 \to 1\) reverses to give a binary choice operator \(\text{or}^{op} : 1 \to 2\). Thus, a comodel \(C\) gives rise to an arrow \(C1 \to 2 \cdot C1\) which consumes binary \(\text{or}\) branches by indicating (via the elements of 2) a choice of ‘left or right’. The sketch for commutativity (in Diagram 2.2) reverses to give a nonsensical constraint on \(\text{or}^{op}\) shown below; whichever ‘left or right’ choice it makes, it makes the same choice even if we swap ‘left’ and ‘right’. The only reasonable way to achieve this is for the comodel carrier \(C1\) to be trivial, i.e. the initial object 0.

\[
\begin{array}{c}
2 \\
\text{or}^{op}
\end{array}
\xymatrix{2 & 2 \\
\text{inr}_1 + \text{inl}_1 & 1
}
\]

Similarly, the associativity and idempotence constraints become unreasonable when reversed. The latter essentially asserts that the comodel never changes state: it always makes the same choice of ‘left’ or ‘right’. One may choose only to consider comodels of the theory with a binary \(\text{or}\) operator and no equations, or equivalently \(\Delta^{co}\)-coalgebras where \(\Delta^{co}X = X + X\).

In analogy to free models, one may describe cofree comodels in terms of an adjunction, where the forgetful functor \(U : \text{Comod}(\mathcal{L}, \mathcal{C}) \to \mathcal{C}\) maps a comodel \(C\) to its carrier \(C1\), and natural transformations between comodels \(\epsilon : C \Rightarrow E\) to their \(1\)-components \(\epsilon_1 : C1 \to E1\).

**Theorem 2.3.7.** [PS04] For any Lawvere theory \(\mathcal{L}\) and locally countably presentable category \(\mathcal{C}\), the forgetful functor \(U : \text{Comod}(\mathcal{L}, \mathcal{C})\) has a right adjoint \(G : \mathcal{C} \to \text{Comod}(\mathcal{L}, \mathcal{C})\).
The right adjoint $G$ maps an object $X$ to a comodel $GX$ which may be thought of as the cofree comodel generated by elements of $X$. In particular, it preserves limits, and hence the final object; thus we are guaranteed the existence of a final comodel $G1$, which intuitively provides a ‘minimal implementation’ of the effects.

For instance, in the case of user I/O, the absence of equational constraints means that comodels coincide with $\Theta^\omega$-coalgebras (as defined in Section 2.3.3); and so the final comodel is in fact the final $\Theta^\omega$-coalgebra (a collection of trees describing all possible interaction paths of the user with a program). In particular, it is shown in [PP08] that the final comodel $C$ for the theory of global store (with a finite set of locations $L$) has carrier $C1$ given by the canonical notion of store, $S = N^L$. Each write effect $wr_{x,n} : 1 \rightarrow 1$ is mapped by the comodel to a function $\text{upd} : S \rightarrow S$ which implements the effect in the natural way, viz. $s \mapsto s[x \mapsto n]$. Similarly, a read effect $rd_x : \omega \rightarrow 1$ gives rise to a function $\text{lku} : S \rightarrow N \cdot S$ which ‘returns’ the value of the variable $x$, along with the resulting, unaltered store – viz. $s \mapsto (s(x), s)$. We will refer to this comodel in our semantics for the \textbf{While} and \textbf{NDWhile} languages in Chapter 4.
Chapter 3

MOS in a Mixed Kleisli Category for Syntactic Effects

In this chapter, we describe the problems that arise in applying Turi and Plotkin’s mathematical operational semantics (MOS) framework to programming languages directly: these are the unconventional nature of the operational specifications, and the excessive fine-grainedness of the semantic domain given by a final coalgebra. We introduce an adaptation of the framework which addresses this problem, by accumulating effects while a program executes. Formally, this is achieved by describing program behaviour in a Kleisli category, and program syntax in the underlying category.

This chapter draws on the paper [ASP11], with one key difference: we work in the category $\mathbf{Cpo}_{\leq}$, rather than $\mathbf{Set}$, to introduce the required order structure more elegantly. The technical details are adapted from the follow-up paper, [ASP13]. The introduction of order structure in this chapter will also facilitate the introduction of Lawvere theories in the next chapter.

3.1 Applying MOS To Programming Languages Directly

As outlined in the previous chapter, in applying MOS to the semantics of a programming language, one must specify its syntax and behaviour functors, and an operational specification, in the form of a natural transformation. Initially, we will illustrate this process with two running example languages – the While language, incorporating a global store of natural number-valued variables, indexed by a set of locations $L$; and the fragment of the language without these variables, which we call SWhile, or ‘stateless While’. The language syntax is defined below, and is expressed in terms of multiple sorts, given by a set $S$, to ensure syntax terms are well-typed. There are three sorts: numeric expressions $N$, boolean expressions $E$, and command
Definition 3.1.1. The language $S\text{While}$ has the following syntax, where $n$ ranges over natural numbers $\mathbb{N}$, and $b$ the set $\mathbb{B} = \{\text{true}, \text{false}\}$.

\[
N ::= n \mid N + N \mid N \times N \mid +_n(N) \mid *_n(N) \\
E ::= b \mid N == N \mid N \leq N \mid =_n(N) \mid \leq_n(N) \mid \neg E \mid E \land E \\
P ::= \text{skip} \mid P ; P \mid \text{while}(E)\{P\} \mid \text{if}(E)\{P\} \text{else}\{P\}
\]

Given a set of variables $L$, the language $\text{While}$ extends the numeric expression syntax $N$ with terms $x$ for all $x \in L$, and also introduces commands $x=N$.

In addition to standard operations such as addition $+$ and multiplication $\times$ of numeric expressions, the syntax of $S\text{While}$ includes auxiliary commands, such as $+_n$, which adds a pre-calculated total $n$ to a numeric expression, and similarly for $*_n$; they will become important when we consider operational specifications in Section 3.3.4. Boolean operations include negation $\neg$ and conjunction $\land$, and testing equality $==$ and comparison $\leq$ of numeric expressions; there are corresponding auxiliary operators $=_n$ for testing if an expression is equal to $n$, and similarly for $\leq_n$. Commands include \textit{if} and \textit{while} statements, sequential composition, and an effectless \texttt{skip} command. The language $\text{While}$ introduces a numeric expression $x$ for returning the value of the variable $x$, and a command $x=N$ for updating a variable with the value of a numeric expression.

Multi-sorted Syntax

In a multi-sorted setting, a syntax constructor takes arguments of several sorts, and returns a syntax term of a particular sort. We formalise this information by an algebraic \textit{signature}, as follows. We permit syntax constructors to take countably many arguments, as this will be required for effect-syntax such as $\mathbb{N}$-ary lookups $\text{rd}(x_1, x_2, \ldots)$.

Definition 3.1.2. A (countable $\mathcal{S}$-sorted) signature $\text{Sig}$ over a set $\mathcal{S}$ of sorts is a set of function symbols $\sigma$, each with an associated type given by: an arity $\alpha \leq \omega$; a sequence of sorts $(s_i)_{0 \leq i < \alpha}$ in $\mathcal{S}$; and an output sort, $s_\sigma$. We represent the type of $\sigma$ by writing $\sigma : (s_i)_{0 \leq i < \alpha} \rightarrow s_\sigma$. If arities are clear from the context, we write $\sigma : (s_i) \rightarrow s_\sigma$.

We will show how to represent the action of these syntax constructors functorially on the category $\mathcal{C}_{\mathcal{S}}$, assuming $\mathcal{C}$ has coproducts and countable products. Its objects are given by $\mathcal{S}$-indexed tuples $X = (X_s)_{s \in \mathcal{S}}$ of objects $X_s$ in $\mathcal{C}$, and its arrows $f : X \rightarrow Y$ are $\mathcal{S}$-tuples of arrows $f_s : X_s \rightarrow Y_s$ from $\mathcal{C}$; they are composed componentwise. In the context of functorial syntax,
we may consider an object \( X = (X_s)_{s \in S} \) as a tuple of syntax variables \( X_s \), each considered to have sort \( s \).

In a multi-sorted setting, given a signature \( \text{Sig} \), we can apply syntax constructors \( \sigma : (s_i)_{0 \leq i < \alpha} \to s_\sigma \) to a tuple of sorted variables \( (X_s)_{s \in S} \). In the previous chapter, applying \( n \)-ary constructors \( \sigma \) to a set of variables \( X \) gave terms which were represented by \( n \)-tuples in \( X^n \). By contrast, here the terms \( \sigma((x_i)_{0 \leq i < \alpha}) \) correspond to tuples of variables \( x_i \) drawn from the appropriate collections \( X_i \). Such tuples are contained in a product \( X_{a_1} \times \ldots \times X_{a_n} \) for appropriate sorts \( a_1, \ldots, a_n \).

**Example 3.1.3.** Taking \( \mathcal{C} = \text{Set} \), the signature for \( \text{SWhile} \) has sorts \( S = \{N, E, P\} \), and the type of the \( \text{if} \) syntax constructor is \( (E, P, P) \to P \). Correspondingly, given the collections of syntax variables \( (X_N, X_E, X_P) \), we may construct syntax terms \( \text{if}(x_E) \text{then} \{x_P\} \text{else} \{x'_P\} \), for all \( x_E \in X_E \) and \( x_P, x'_P \in X_P \). These terms correspond to tuples in \( X_E \times X_P^2 \) – i.e. sequences in \( (X_E, X_P, X_P) \) – and they are considered to have sort \( P \). Similarly, from these syntax variables, we can represent \textbf{while} loops by tuples in \( X_E \times X_P \), sequential compositions by pairs \( X_P \times X_P \), and the \textbf{skip} command by a singleton set 1.

The numeric syntax terms are as follows. The natural-number constants \( n \) can be represented by \( \mathbb{N} \); the binary addition and multiplication operations, applied to variables \( X_N \), are represented by pairs in \( X_N^2 \); and finally, each unary operator \( +_n \) and \( \times_n \) directly by \( X_N \) – and so collectively by \( \mathbb{N} \times X_N \). We represent the boolean operators in a similar vein.

As before, each syntax constructor \( \sigma \) applied to syntax variables \( (X_s)_{s \in S} \) gives a collection \( C_\sigma \) of tuples, and we combine these collections via their coproduct. The difference is that now the terms, or tuples, may be of different sorts: the constructed terms of type \( s \in S \) are given by taking the coproduct of the collections \( C_\sigma \) whose terms are of sort \( s_\sigma = s \).

**Example 3.1.4.** Given a tuple \( (X_N, X_E, X_P) \) of variables (assumed to be of numeric, boolean, and command sort respectively), one may describe a single application of the command-type syntax constructors of \( \text{SWhile} \) by the elements of the coproduct \( 1 + X_P \times X_P + X_E \times X_P + X_E \times X_P^2 \) – corresponding respectively to \textbf{skip}, sequential compositions \( p ; q \), \textbf{if} statements, and \textbf{while} loops. Similarly, applications of the syntax constructors of numeric type may be represented by the coproduct \( \mathbb{N} + X_N^2 + X_N^2 + \mathbb{N} \times X_N + \mathbb{N} \times X_N \) – the elements of its components correspond to naturals \( n \), addition \( x + y \) and multiplication \( x \times y \), and the auxiliary commands \( +_n(x), \times_n(x) \) for each \( n \in \mathbb{N} \).

For \textbf{While}, we introduce variable updates \( x = N \), for all \( x \in L \), by adding \( L \times X_N \) to the first coproduct shown above, for syntax terms of command type; and variable lookups \( x \), by adding constants \( L \) to the second coproduct above (so that they are considered to have numeric type).
Each coproduct of sort \( s \) may be described by applying a suitable functor \( \Sigma_s : C^S \to C \) to a collection of variables \( X = (X_s)_{s \in S} \) given by an object of \( C^S \). Thus \( \Sigma_s X \) represents the terms of sort \( s \) which may be formed from variables \( X \). Taking the product of the functors \( \Sigma_s \) gives an endofunctor \( \Sigma = (\Sigma_s)_{s \in S} : C^S \to C^S \) which constructs syntax terms of every sort simultaneously; applied to an object \( X \) of \( C^S \), it gives an object \( \Sigma X \) whose \( s \)-component \( (\Sigma X)_s \) describes the terms of sort \( s \), given by \( \Sigma_s X \).

As before, an arrow \( f : X \to Y \) in \( C^S \) may be thought of as relabelling the variables \( X_s \) to \( Y_s \) for all sorts \( s \). The corresponding arrow \( \Sigma_s f : \Sigma_s X \to \Sigma_s Y \) relabels the \( s \)-sorted terms constructed from variables \( X \), into terms over \( Y \). The arrow \( \Sigma f : \Sigma X \to \Sigma Y \) performs this relabelling at all sorts simultaneously.

**Example 3.1.5.** For \textbf{SWhile}, we describe the action of syntax constructors by an endofunctor \( \Sigma = (\Sigma_s)_{s \in \{N,E,P\}} : \text{Set}^3 \to \text{Set}^3 \). The functor representing the command-type constructors of Definition 3.1.1, in the order given there, is \( \Sigma_P(X_N, X_E, X_P) = 1 + X_P \times X_P + X_E \times X_P + X_E \times X_P^2 \), and similarly \( \Sigma_N, \Sigma_E \) for the other sorts. Finally, by taking the product over \( s \in S \), we may define

\[
\Sigma(X_N, X_E, X_P) = \begin{pmatrix}
\Sigma_N(X_N, X_E, X_P) \\
\Sigma_E(X_N, X_E, X_P) \\
\Sigma_P(X_N, X_E, X_P)
\end{pmatrix}
= \begin{pmatrix}
\mathbb{N} + X_P^2 + X_P^3 + \mathbb{N} \times X_N + \mathbb{N} \times X_N \\
\mathbb{N} \times X_N + X_N \times X_N + \mathbb{N} \times X_N + \mathbb{N} \times X_N + X_E + X_E \times X_E \\
1 + X_P \times X_P + X_E \times X_P + X_E \times X_P^2
\end{pmatrix}
\]

With this functorial representation of syntax constructors in \( C^S \), we may re-use the formal methods of the previous chapter to construct the syntax terms of the language. As before, to build closed terms (containing no variables), one must start with an empty collection of variables for each sort, and apply the syntax constructors repeatedly. In the single-sorted setting, this was formally represented by the initial \( \Sigma \)-algebra, constructed via the initial sequence, by repeatedly applying \( \Sigma \) to the initial object \( 0 \) of \( C \) (e.g. the empty set in \( \text{Set} \)), and taking a colimit. In this setting, we start with \( S \) copies of \( 0 \), one for each sort; this is the initial object in \( C^S \) and so we also call it \( 0 \). Now we may apply \( \Sigma : C^S \to C^S \) repeatedly to \( 0 \) to build the syntax terms at every sort simultaneously; the collection of all terms is again given by an initial \( \Sigma \)-algebra \( T0 \). As before, one may instead start with a collection of syntax variables \( X = (X_s)_{s \in S} \), and build syntax terms \( TX \) over \( X \) given by the initial \( (\Sigma + X) \)-algebra; this gives rise to a free \( \Sigma \)-algebra functor \( T \).

**Remark 3.1.6.** In our running examples, existence of these initial algebras will be assured by working in the base category \( C = \text{Cpo}_{\downarrow} \), in Section 3.3.3. However, we give some remarks
about ensuring existence in a more general setting. Remarks 2.1.8 and 2.1.10 showed that
initial algebras exist if the category $C^S$ has $\omega$-colimits, and that the functor $\Sigma$ preserves them.
Concerning the first point, note that colimits in $C^S$ are given pointwise, assuming $C$ has those
colimits. In particular, the $\omega$-chain in $C^S$ of the initial $\Sigma$-sequence $0 \to \Sigma_0 \to \ldots$ 
consists of $s$ diagrams (in $C$), which are also $\omega$-chains: $0 \to \Sigma_s0 \to \Sigma_s\Sigma_0 \to \ldots$. If $C$ has colimits of 
$\omega$-chains, then so does $C^S$; under this assumption, the colimit of the initial $\Sigma$-sequence is then 
the product of the colimits in $C$ for all $s \in S$.

As for $\Sigma$ preserving $\omega$-colimits, for the same reasons it is enough to show that each component $\Sigma_s : C^S \to C$ preserves them. Assuming that $\Sigma_s$ is finitely polynomial, it may be described 
by compositions of arbitrary coproduct functors $+: C^n \to C$, and finite product functors $\times$. The former preserve colimits as they are left adjoints to diagonal functors $C \to C^n$; the latter preserves $\omega$-colimits by Theorem 1 in Section IX.2 of [ML71].

### 3.1.1 Functorial Behaviour

As before, we may characterise an operational model of a programming language as an $B$-
coalgebra $(X, \gamma)$, for a suitable endofunctor $B$ characterising transition behaviour. In the 
setting of $C^S$, the state-space $X$ is divided into components $X_s$ for each sort $s$; the transition 
structure $\gamma : X \to BX$ consists of an arrow $X_s \to (BX)_s$ for every $s \in S$, describing the transition behaviour of states of that sort. We consider our example languages in turn.

- **SWhile**. We assume the $s$-sorted states $x_s$ of the transition system are given by a collection $X_s$; in particular, for the operational model of SWhile, one would take $X_s$ to be the collection of program terms $(T0)_s$ of sort $s$. Execution proceeds via unlabelled transitions $x_s \to x'_s$, and may terminate $x_s \to v$ with a return value $v$, from a suitable collection $V_s$. (We distinguish return values $v$ from syntactic values $v$ by underlining.) More specifically, we take $V_N = \mathbb{N}, V_E = \mathbb{B}$ and $V_P = 1$ (a void return value $\ast$ for command expressions). An example execution might be as follows – although the operational models of the next section will differ slightly, as they require auxiliary functions.

$$1 + 2 + 4 \to 3 + 4 \to \top \quad (3 = 3) \land \text{true} \to \text{true} \land \text{true} \to \text{true}$$

To each state $x_s$ in $X_s$, such transitions assign either a new state $x'_s$ in $X_s$ (such as $3 + 4$), or a terminal value $\top$ in $V_s$ (such as $\top$). These possibilities may be combined and represented by the components of a coproduct $V_s + X_s$, so that the behaviour of the state $x_s$ is either $\text{inl}(\top)$ or $\text{inr}(x'_s)$ respectively; but as before, for readability we will often omit these injections.
The assignment of behaviours \( \text{inl}(\gamma), \text{inr}(x') \) to states \( x \) may be represented by an arrow \( \gamma_s : X_s \rightarrow V_s + X_s \), or equivalently \( X_s \rightarrow B_s X_s \), where we define the functor \( B_s : \mathcal{C} \rightarrow \mathcal{C} \) by \( B_s Y = V_s + Y \); this expresses the transitions of the \( s \)-sorted states \( X_s \) as a \( B_s \)-coalgebra \( (X_s, \gamma_s) \).

We may then combine the arrows \( \gamma_s \) defining transition structures at each sort, giving a single arrow \( \gamma \) in \( \mathcal{S} \); this is the structure of a \( \mathcal{B} \)-coalgebra in \( \mathcal{S} \) where we define \( B : \mathcal{S} \rightarrow \mathcal{S} \) with \( (B X)_s = B_s X_s \) – or equivalently \( B X = V + X \), where the coproduct is componentwise in \( \mathcal{S} \). Then the collection of arrows \( \gamma_s : X_s \rightarrow V_s + X_s \) is equivalent to a single arrow \( \gamma : X \rightarrow B X \) in \( \mathcal{S} \), making the operational model for \( \text{SWhile} \) into a \( \mathcal{B} \)-coalgebra.

- \textbf{While}. The operational model of this language incorporates global store, and hence may be described in terms of the side-effect monad of Section 2.3.1, where the stores are given by \( S = \mathbb{N}^L \). One again has a state-space \( X_s \) at each sort \( s \), which in the operational model would be taken to be the \( s \)-sorted terms \( (T0)_s \) of the language. However, execution now proceeds in terms of \textit{configurations} \( \langle x_s, c \rangle \) of an \( s \)-sorted term \( x_s \) and a store \( c \in S \); an evaluation step either produces a new configuration \( \langle x_s, c \rangle \rightarrow \langle x'_s, c' \rangle \), or a terminal value: \( \langle x_s, c \rangle \rightarrow \langle v, c' \rangle \), where \( v \) is again drawn from a collection \( V_s \). We may take the same values \( V_s \) as we did for \textbf{SWhile}. Typical transitions might be as follows, where \( c(y) \) is the value of the variable \( y \) in store \( c \), and \( c[y \mapsto n] \) is the store \( c \) where \( y \) has been reassigned to \( n \).

\[
\langle y + 5, c \rangle \rightarrow \langle c(y) + 5, c \rangle \rightarrow \langle c(y) + 5, c \rangle \quad \langle y = 3, c \rangle \rightarrow \langle *, c[y \mapsto 3] \rangle
\]

This transition data can be represented by a function \( \gamma'_s : (X_s \times S) \rightarrow ((V_s + X_s) \times S) \); as for \textbf{SWhile}, we distinguish non-terminal transitions (e.g. the first transition above) from terminal ones (e.g. the others) by considering them as separate components \( \text{inl}(\gamma), \text{inr}(x') \) of a coproduct \( V_s + X_s \).

Assuming \( \mathcal{C} \) is closed (c.f. Definition 2.1.16), the arrow \( \gamma'_s \) may be curried, and re-expressed as an arrow \( \gamma_s : X_s \rightarrow ((V_s + X_s) \times S) \). Whereas the arrow \( \gamma'_s \) assigns a transition behaviour to each \textit{configuration} \( \langle x_s, c \rangle \), the arrow \( \gamma_s \) assigns a behaviour to each \textit{program-state} \( x_s \) of the transition system – a function which takes the state \( c \) as a parameter. For instance, the program \( y = 3 \) would be assigned by \( \gamma_s \) to the following function. (We have explicitly included the formal injection \( \text{inl}(\gamma) \) of the return value \( * \) into the left-hand component 1 of the \( (P\text{-sorted}) \) coproduct \( B_p X_p = 1 + X_p \).

\[
\gamma_s(y = 3) = \lambda c. (\text{inl}(\gamma), c[y \mapsto 3])
\]

To express this transition behaviour coalgebraically, we define \( M_0 \) to be the side-effect monad \( M_0 Y = (Y \times S)^S \) (Section 2.3.1) on the base category \( \mathcal{C} \), and \( B_s Y = V_s + Y \) as above, so that the \( s \)-sorted transition function \( \gamma_s : X_s \rightarrow (B_s X_s \times S)^S \) becomes the struc-
3.1. Applying MOS To Programming Languages Directly

3.1. Applying MOS To Programming Languages Directly

M-bcoalgebra. We may then define functors \(M, B : C^S \to C^S\) componentwise as above – i.e. \((MY)_s = M_0Y_s\) and \((BY)_s = B_sY_s\) – so that the arrows \(\gamma_s\) are equivalent to a single arrow \(\gamma : X \to MBX\) in \(C^S\); the operational model for While is then framed as an \(MB\)-coalgebra.

In the same vein, other operational models – such as While with user I/O and/or non-determinism – may be described as \(MB\)-coalgebras, for various computational monads \(M\). This includes SWhile, by taking \(M\) to be the identity monad.

3.1.2 Operational Rules and Models

As described in Section 2.1.8, we may derive an operational model for a language by supplying an abstract OS specification; taking the behaviour functor to be the composition \(MB\), these are natural transformations \(\epsilon_X : \Sigma(X \times MBX) \to MBTX\). We will see that although such specifications are feasible for simple languages like SWhile, in the case of While they bear little resemblance to standard operational specifications.

SWhile. We take \(M\) to be the identity monad and omit it, and also \(BX = V + X\) where \(V = (\mathbb{N}, \mathbb{B}, 1)\) as before. Abstract OS specifications are then natural transformations \(\epsilon_X : \Sigma(X \times (V + X)) \to V + TX\), which may be interpreted as follows. Given a (well-typed) syntax term \(\sigma((x_i)_{i \in I})\), its arguments \(x_i\) have behaviours given either by successor states \(x'_i\) or terminal values \(v_i\) drawn from \(V_s\); the transition behaviour of \(\sigma((x_i)_{i \in I})\) specified by \(\epsilon_X\) is allowed to vary depending on which arguments do or do not terminate, as well as any terminal values produced. If the transition produces a syntax term, \(\sigma((x_i)_{i \in I}) \to t\), the term \(t\) may contain any of the arguments \(x_i\), in addition to any of their successors \(x'_i\).

This information may be represented straightforwardly by SOS rules. Rather than formalise the resulting rule format, we illustrate the generality of such specifications with examples of definable behaviours. One may specify a boolean term \(\sigma(x)\) which determines whether or not the argument \(x\) (e.g. of numeric type) terminates in the next step, given by the following rules.

\[
\begin{align*}
x \to x' & \quad \sigma(x) \to \text{false} \\
x \to n & \quad \sigma(x) \to \text{true}
\end{align*}
\]

More generally, a term \(\sigma(x_1, \ldots, x_n)\) might return the number of subterms \(x_i\) which terminate immediately. More convoluted still, if all the sub-terms \(x_i\) are assumed to be numeric type, the term could instead return the sum of whatever return values they would produce if executed for one step; or it could transition to the first argument \(x_i\) which does not terminate on the next step (and return \(0\) if all arguments terminate).
Finally, we may specify the intended behaviour of $\textbf{SWhile}$ syntax constructors by SOS rules, such as the following (and other familiar rules for the remaining operators $\ast,-,=,\leq,\wedge$). This includes specifications of the auxiliary functions $\ast_n,=\ast_n,\leq_n$.

\[
\begin{array}{cccccc}
 n \rightarrow n & b \rightarrow b & \text{skip} \rightarrow \bot & u \rightarrow u' & u + v \rightarrow u' + v & u \rightarrow n \\
 e \rightarrow e' & e \rightarrow \text{false} & e \rightarrow \text{true} & u + v \rightarrow +_n(v) & +_n(v) \rightarrow +_n(v') & +_n(v) \rightarrow n + m \\
 e \wedge f \rightarrow e' \wedge f & e \wedge f \rightarrow \text{false} & e \rightarrow e' & u \rightarrow u' & u \rightarrow n \\
 & u \rightarrow u' & u \rightarrow n & u \rightarrow u' & (\text{etc.}) \\
 & u \rightarrow u' & u \rightarrow n & u \rightarrow u' & (\text{etc.}) \\
\end{array}
\]

\[
\begin{array}{c}
\text{if} (e) \text{then}\{p\} \text{else}\{q\} \rightarrow \text{if} (e') \text{then}\{p\} \text{else}\{q\} \\
\text{while} (e) \text{do}\{p\} \rightarrow \text{if} (e) \text{then}\{p\} \text{while} (e) \text{do}\{p\} \text{else}\{\text{skip}\}
\end{array}
\]

To illustrate the action of our earlier examples, whose operational semantics are now as follows.

\[
1 + 2 + 4 \rightarrow +_3 (4) \rightarrow \text{I} \quad (3 = 3) \land \text{true} \rightarrow (=_3 (3)) \land \text{true} \rightarrow \text{true} \rightarrow \text{true}
\]

Lastly, we give a partial definition of a map $\epsilon_X$ showing how the above information is represented concretely. (We abbreviate $\text{if} (e) \text{then}\{p\} \text{else}\{q\}$ to $\text{if}(e,p,q)$, and similarly $\text{while}(e,p)$.)

\[
\begin{align*}
\ast_n ((u,b_u),(v,b_v)) & \quad \rightarrow \quad \text{Cases}\{ b_u = n \in \mathbb{N} : +_n(v), b_u = u' \in N : (u' + v) \} \\
\ast_n ((v,b_v)) & \quad \rightarrow \quad \text{Cases}\{ b_v = m \in \mathbb{N} : (n + m), b_v = v' \in N : +_n(v') \} \\
\text{if} ((e,b_e),(p,b_p),(q,b_q)) & \quad \rightarrow \quad \text{Cases}\{ b_e = \text{false} \in \mathbb{B} : (q), b_e = \text{true} \in \mathbb{B} : (p), \\
& \quad \text{else}\{ (e',p,q)) \} \\
\text{while} ((e,b_e),(p,b_p)) & \quad \rightarrow \text{if}(e, (p;\text{while}(e,p), \text{skip})
\end{align*}
\]

Each of these assignments may be represented categorically, in terms of (co)products and functions depending on values.

**Example 3.1.7.** We illustrate how such specifications are defined categorically, with the example of $\text{if}(e,p_1,p_2)$ statements, containing boolean expressions $e \in E$ and commands $p_1, p_2 \in P$. They are represented syntactically by tuples in $E \times P^2$; this is a component of the coproduct defining $\Sigma$. The type of the abstract OS is $\Sigma(X \times BX) \rightarrow BTX$; accordingly, the fragment of $\epsilon_X$ specifying if statements has type $(E \times BE) \times (P \times BP)^2 \rightarrow BTX$; as input, it takes the three arguments of the if statements (in $E, P, P$ respectively), as well as their transition behaviours $BE,BP,BP$.

To define this natural transformation and specify if statements, note that we only require the behaviour of the boolean expression $e$ to deduce the behaviour of $\text{if}(e,p_1,p_2)$. Thus, we may discard the argument $e$ and the transition behaviours of $p_i$, as shown in the first step of the definition below. We are left with the behaviour of $e$ – either a boolean return value in $\mathbb{B}$, or a new expression $e'$ – and the arguments $p_i$. These projections form the first
3.1. Applying MOS To Programming Languages Directly

map in our definition of $\epsilon_X$, shown below; the second map is the distributivity isomorphism

$$\text{dist}_{X,Y,Z} : X \times (Y + Z) \cong X \times Y + X \times Z.$$ 

These maps isolate the information we need about the behaviour of the arguments $e,p_1,p_2$ of the if statement: if the boolean expression $e$ returns a value $b$, then these maps return a tuple $(b,p_1,p_2)$ in the first coproduct component $\mathbb{B} \times P^2$, containing the return value $b \in \mathbb{B}$ and the other arguments $p,q$ of the if statement. On the other hand, if $e$ has a transition to $e'$, the maps return a tuple $(e',p_1,p_2)$ in the other component $E \times P^2$, describing this successor state $e'$ and the arguments $p_1,p_2$.

We must now use this information to specify the behaviour of the if statement according to the operational rules, by mapping each tuple $(b,p,q)$ or $(e',p,q)$ to a suitable transition behaviour in $(BTX)_P$ – either the unique ’void’ return value $\ast \in V_P$ for command-type terms, or a successor term $t$. The operational rules imply that we should map a tuple $(\text{true},p_1,p_2)$ to the term $p_1$, and a tuple $(\text{false},p_1,p_2)$ to the term $p_2$, corresponding to the first two derivation rules below; a tuple $(e',p_1,p_2)$ should be mapped to the term $\text{if } (e') \text{ then } \{p_1\} \text{ else } \{p_2\}$, given by the third rule.

$$
\begin{align*}
  e &\rightarrow \text{true} & e &\rightarrow \text{false} \\
  \text{if} (e) \text{ then } \{p_1\} \text{ else } \{p_2\} &\rightarrow p_1 & \text{if} (e) \text{ then } \{p_1\} \text{ else } \{p_2\} &\rightarrow p_2 \\
  e &\rightarrow e' & \text{if} (e) \text{ then } \{p_1\} \text{ else } \{p_2\} &\rightarrow \text{if} (e') \text{ then } \{p_1\} \text{ else } \{p_2\}
\end{align*}
$$

More formally, to handle the first two cases, we must construe the variables $p_1,p_2 \in X_P$ as syntax terms $(TX)_P$ by using the (command- or $P$-component of the) monad unit $(\eta_X)_P : X_P \rightarrow (TX)_P$. Moreover, as these are non-terminal transitions, they should be injected (via inj) into the right-hand component of the behaviour functor $BX = V + X$.

We now formalise these considerations by defining suitable maps $g_1 : \mathbb{B} \times P^2 \rightarrow (BTX)_P$ and $g_2 : E \times P^2 \rightarrow (BTX)_P$, allowing us to define an abstract OS specification by the composition

$$
\mathbb{B} \times P^2 + E \times P^2 \xrightarrow{[g_1,g_2]} (BTX)_P = 1 + (TX)_P
$$

The first map $g_1 : \mathbb{B} \times P^2 \rightarrow (BTX)_P$ has to define the behaviour of the if statement when $e$ terminates with a value in $\mathbb{B}$. If it terminates with value true, the behaviour of the if statement should be if $(e,p_1,p_2) \rightarrow p_1$; hence, we must return the first argument $p_1$, by applying a projection $\pi_1$ to the product $P^2$. If $e$ returns false, we instead return the second argument
correspond to components of a coproduct: this means applying distributivity once more, and isomorphisms $\alpha_X : 1 \times X \cong X$ as shown below, followed by the appropriate projections.

$$
(1 + 1) \times P^2 \xrightarrow{\text{dist}_{1,1}.P^2} (1 \times P^2) + (1 \times P^2) \xrightarrow{\alpha_{P^2} + \alpha_{P^2}} P^2 + P^2 \xrightarrow{[\pi_2, \pi_1]} P
$$

Having isolated the appropriate successor term $p_1$ in $P$, we construe it as a syntax term $(TX)_P$, by applying the $P$-component of the monad unit $\eta_X$. Finally, as these behaviours are non-terminal transitions, we must inject $p_1$ into the second component of the codomain $1 + (TX)_P$, by applying $\text{inr}_{(TX)_P}$. Thus, we define $g_1 = \text{inr}_{(TX)_P} \circ (\eta_X)_P \circ [\pi_2, \pi_1] \circ (\alpha_{P^2} + \alpha_{P^2}) \circ \text{dist}_{1,1}.P^2$.

We now deal with the other component of the coproduct, $E \times P^2$, describing the arguments $e', p_1, p_2$ when $e$ has a transition $e \to e'$. We construe such a tuple as a syntax term $\text{if}(e', p_1, p_2)$ statement by injecting it into the appropriate component of $(\Sigma X)_P$ – the fourth one, from Example 3.1.5; we call the injection $\text{in4}$. For bureaucratic reasons, we must treat this as an arbitrary-depth syntax term in $(TX)_P$, through the $(P$-component of the) sequence of compositions $\Sigma X \xrightarrow{\Sigma \eta_X} \Sigma TX \xrightarrow{\psi_X} TX$ (where $\psi_X$ is the $\Sigma$-algebra structure of the free $\Sigma$-algebra $TX$, as in Section 2.1.3). Finally, as this is a non-terminal transition, we inject the term into the second component of $1 + (TX)_P$. This leads to the definition $g_2 = \text{inr}_{(TX)_P} \circ (\psi_X)_P \circ (\eta_X)_P \circ \text{in4}$.

**While.** Assuming $M$ is given componentwise by the side-effect monad (which we called $M_0$ above), and taking the same functor $BX = V + X$ as for SWhile, the specifications for $MB$-coalgebras are natural transformations $\epsilon_X : \Sigma(X \times ((V + X) \times S)^S) \to ((V + TX) \times S)^S$ giving rise to an extremely general rule format. The behaviour of a syntax term $\sigma_{\{x_i\}_{i \in I}}$ is allowed to depend on the ‘curried’ versions of the sub-terms’ behaviours: functions of the initial store, of type $((V + X) \times S)^S$. In other words, it is allowed to inspect how each argument $x_i$ would behave when paired with any store $c$, i.e. whether $\langle x_i, c \rangle \to \langle x'_i, c' \rangle$ or $\langle x_i, c \rangle \to \langle v_i, c' \rangle$ (for some $c', x'_i$ or $v_i$). The behaviour of a configuration $\langle \sigma_{\{x_i\}_{i \in I}}, c \rangle$ can then depend on all this information, in addition to the store $c$.

Again, to illustrate the generality of this format, let $y \in L$ be a variable in the store. One could specify a unary term $\sigma_y(p)$ which, for all initial stores $c$, inspects the transitions of the term $p$: either $\langle p, c \rangle \to \langle p', c' \rangle$ or $\langle p, c \rangle \to \langle v, c' \rangle$. Then it may return the minimum value of the variable $y$ in any of the resulting stores $c'$, i.e. $\text{min}_{c \in L}(c'(y))$.

It is not feasible to give a syntactic presentation of this format; however, we sketch how a suitable restriction can give a syntactic SOS-style presentation. First, note that many of the standard SOS rules for While involve premises such as $\langle x, c \rangle \to \langle x', c' \rangle$, where $c$ and $c'$ are store-variables representing arbitrary stores. The rule implicitly refers to a store-manipulating function $f : S \to S$ with $f(c) = c'$, which is re-used in rule conclusions of the form $\langle \sigma_{\{x_i\}_{i \in I}}, c \rangle \to$
3.1. Applying MOS To Programming Languages Directly

\[ \langle \sigma'((x'_i)_{i \in I}), c' \rangle \]. The intended meaning is that when evaluating \( \sigma((x_i)_{i \in I}) \), the effect on the store \( c \) is given by applying the same function \( f \). (There is a similar relationship between \( c \) and the transition behaviour of \( x \), and we omit the details.)

By contrast, conclusions such as \( \langle x = n, c \rangle \rightarrow \langle \ast, c'[x \mapsto n] \rangle \) imply a fixed store-manipulation function – such as the function \( f : c \mapsto c'[x \mapsto n] \) which updates \( x \) to \( n \), or the identity function \( f(c) = c \). As another example, a rule might imply the composition of state-manipulations, such as

\[
\begin{align*}
\langle x, c \rangle &\rightarrow \langle x', c' \rangle, \quad \langle y, c' \rangle \rightarrow \langle y', c'' \rangle \\
\langle \sigma(x, y), c \rangle &\rightarrow \langle \sigma(x', y'), c'' \rangle
\end{align*}
\]

We may thus frame operational rules in terms of the store-manipulations of sub-terms; these are essentially the curried form of coalgebraic behaviour we have considered. Syntactically, one could achieve this by defining a rule format restricting the placement of both program terms and store-variables, to ensure a consistent dependency of the final store (and new sub-terms) on the initial store. We will do this in a restricted manner in Section 4.2.1, giving rise to what we call the EIC1 format (Definition 4.2.4).

For now, we simply point out that most of the operational rules for While may be obtained from those for SWhile in a natural way, by replacing premises \( x_i \rightarrow b \) with \( \langle x_i, c \rangle \rightarrow \langle b, c'_i \rangle \) (where \( b \) is a term \( t((x_i)_{i \in I}) \) or terminal value \( v \)), and similarly, replacing conclusions \( \sigma((x_i)_{i \in I}) \rightarrow b \) with \( \langle \sigma((x_i)_{i \in I}), c \rangle \rightarrow \langle b, d \rangle \), where the final store \( d \) is either a function \( f(c) \) of \( c \), or one of the stores \( c'_i \) for some \( i \). This choice of a function \( f(c) \) or a store \( c'_i \) respectively implies a fixed manipulation of the store, or the store-manipulation carried out by the argument \( x_i \); and the resulting behaviour \( b \) only uses information given by those store-manipulations. Adding the standard rules for lookup and update to SWhile, we obtain rules such as the following.

\[
\begin{align*}
\langle n, c \rangle &\rightarrow \langle n, c \rangle, \quad \langle u + v, c \rangle \rightarrow \langle u' + v, c' \rangle \quad \langle x = u, c \rangle \rightarrow \langle \ast, c'[x \mapsto n] \rangle \\
\langle x, c \rangle &\rightarrow \langle c(x), c \rangle
\end{align*}
\]

The rule conclusions contain information which may be obtained from the stateful behaviour of the sub-terms \( u \). This allows us to derive an explicit natural transformation \( \epsilon_X \) relating the behaviour of a term to the (curried) behaviour of its sub-terms, although the details are quite involved. We will give an example of such a definition when we introduce the Concrete Evaluation-In-Context 1 format, in Section 4.2.1.

The example of the While language illustrates a problem with abstract OS specifications: they may instantiate to rule formats which bear little resemblance to the standard operational semantics of stateful programs, and it is hard to describe such specifications explicitly. The impracticality of abstract OS specifications is one reason why we will need restrictions on abstract OS specifications, as introduced in Section 3.3.4 and then Section 4.2.1.

Nonetheless, because of their extreme generality, it is possible in principle to specify a wide
variety of behaviours through abstract OS specifications $\epsilon$. As described in Section 2.1.26, this allows us to apply structural recursion over the empty set of generators 0, with trivial coalgebra structure $?_{MB0}: 0 \rightarrow MB0$, to induce an operational model $om := T^*(?_{MB0}) : T0 \rightarrow MBT0$ for the language.

### 3.1.3 Behavioural Equivalence and the Final Kleisli Coalgebra

In addition to the difficulty of defining abstract OS specifications, another problem arises when applying Turi and Plotkin’s framework to programming languages considered as MB-coalgebras: the final MB-coalgebra $(D, \zeta)$ is an overly fine-grained semantic domain. Recall that program behaviour, as given by an operational model, is characterised by the morphism into the final coalgebra. This results in an overly fine-grained characterisation of program behaviour, distinguishing behaviours which should be considered equivalent.

First, one has to guarantee the existence of the final MB-coalgebra; at the end of this section, we make some comments about this point. However, we proceed under the assumption that it exists, and that the final coalgebra is given as a limit of the final MB-sequence $1 \leftarrow MB1 \leftarrow MBMB1 \leftarrow \cdots$ up to $\omega$.

The first few objects of the sequence give an insight into the characterisation of behaviour by the final coalgebra; intuitively, the $n^{th}$ object $(MB)^n1$ describes the information that can be observed about a program’s behaviour in $n$ steps, where $* \in 1$ plays the role of an arbitrary program state as before. (In a multi-sorted setting, 1 is the final object of $C^S$, given componentwise by the final object of $C$.)

We illustrate with reference to While, where $M$ is given componentwise by the side-effect monad, and $BX = V + X$. For $n = 1$, each element of the object $MB1 = ((V + 1) \times S)^S$ is a function which describes, for each initial store $s$, what can be observed when some program undergoes a single transition step: namely, the new store $s'$, and the information that it has either terminated (returning a value $\text{inl}(\nu)$) or that there has been a transition to some new program state, which we represent by the element $\text{inr}(*)$ of the right component of $V + 1$ at sort $s$. (This transition is different from a ‘void’ return value $\text{inl}(\star)$ – which would be given by a left injection).

**Example 3.1.8.** The program $p_1 : x = 5; x = 1$ – an element of $(T0)p$ – has non-terminal transition behaviour $\langle p_1, c \rangle \rightarrow \langle x = 1, c[x \mapsto 5] \rangle$. In curried form, and with explicit coproduct injections, this transition behaviour is represented by the function shown below-left, of type $(MBT0)p$. The corresponding element of the object $(MB1)p$ abstracts away the successor state $x = 1$, replacing it with $*$; this gives the function below-right, which represents all the observable information about the 1-step behaviour of $p_1$ – its effect on the store, and the fact
that it has not terminated (as indicated by $\text{inr}(\ast)$).

\[ \lambda c. (\text{inr}(x = 1), c[x \mapsto 5]) \quad \lambda c. (\text{inr}(\ast), c[x \mapsto 5]) \]

However, for $n \geq 2$, the object $(MB)^n1$ contains too much information about the behaviour of programs in $n$ steps. Taking $n = 2$, given a state $x$ of an $MB$-coalgebra (such as a term in the operational model for While), its 2-step behaviour is characterised by some function $\phi$, of type $MBMB1 = ((V + ((V + 1) \times S) \times S) \times S)$. When the initial store is $c$, the immediate transition behaviour of $\langle x, c \rangle$ is given by $\phi(c) = (\kappa, c')$. This contains the new store $c'$ after one execution step, and a transition function $\kappa$ which depends on whether $\langle x, c \rangle \rightarrow \langle v, c' \rangle$, or $\langle x, c \rangle \rightarrow \langle x', c' \rangle$. In the former case, we have $\kappa = \text{inl}(v)$, and in the latter case, $\kappa$ is a function $((V + 1) \times S) = MB1$ describing the 1-step behaviour of the new state $x'$. This function tells us the behaviour of the intermediate state $\langle x', d \rangle$ for every store $d$, not just the actual store $c'$ produced by the first transition of $x$.

**Example 3.1.9.** The program $p_1 : (x = 1 ; x = 5)$ has a transition to the successor-state $x = 1$, whose *one-step* behaviour (of type $(MBT0)_P$ again) is shown below; as it terminates, there are no successor-states to abstract away, and its 1-step representation in the final coalgebra (an element of $(MB1)_P$), which we called $\kappa$ above, is essentially the same function.

\[ \kappa = \lambda c. (\text{inl}(x), c[x \mapsto 1]) \]

This describes the observable information (in 1 step) about the successor-state $x = 1$ of $p_1$, and it appears in the characterisation $\phi$ of the *two-step* behaviour of $p_1$. One may obtain $\phi$ by considering the coalgebraic transition behaviour of $p_1$ (shown again below-left), and replacing the successor-state $x_1$ with its 1-step observable information $\kappa$; the result is shown below-right. Note that this function encapsulates the information that $p_1$ updates the store by assigning $x \mapsto 5$ on the first step, in addition to the fact that it assigns $x \mapsto 1$ on the second step.

\[ \lambda c. (\text{inr}(x = 1), c[x \mapsto 5]) \quad \lambda c. (\text{inr}(\lambda c'. (\text{inl}(x), c'[x \mapsto 1])), c[x \mapsto 5]) \]

This means that the final $MB$-coalgebra characterises each program’s behaviour by its interaction with the store *at every execution step*, rather than its overall effect on the store. For instance, we can show that in the standard operational model $(T0, \text{om})$ for While, the two programs $p_1 : (x = 5 ; x = 1)$ and $p_2 : (x = 1 ; x = 1)$ would be sent to different elements $d_1, d_2$ of the final coalgebra $D$ by the coalgebra morphism $\beta_{\text{om}} : T0 \rightarrow D$. One may show that the 2-step behaviours $\phi_1, \phi_2$ exhibited by $d_1, d_2$, of type $MBMB1$ are as shown below, so that the behaviours of $p_1$ and $p_2$ are considered to be different in the final-coalgebra semantics, which is an undesirable situation.
Chapter 3. MOS in a Mixed Kleisli Category for Syntactic Effects

\[ \phi_1 : \lambda c. (\text{inr}(\lambda c'.(\text{inl}(\ast)), c'[x \mapsto 1]), c[x \mapsto 5]) \]
\[ \phi_2 : \lambda c. (\text{inl}(\lambda c'.(\text{inl}(\ast)), c'[x \mapsto 1]), c[x \mapsto 1]) \]

**Remark 3.1.10.** The two-step behaviours \( \phi_1, \phi_2 \) can be obtained explicitly from the (implicit) denotations \( d_1 = \beta_{\text{om}}(p_1) \) and \( d_2 = \beta_{\text{om}}(p_2) \) by post-composing the coalgebra morphism \( \beta_{\text{om}} : T_0 \to D \) with the arrow \( g_2 : D \to MBMB1 \) from the limiting cone over the final MB-sequence (which characterises the limit as the final MB-coalgebra). To prove this is the case, one can use the fact that the coalgebra morphism \( \beta_{\text{om}} \) is a cone morphism from the cone induced by the operational model \( \text{om} \) into the limiting cone over the final sequence. The map \( f_2 : T_0 \to MBMB1 \) from the first cone is given by the composition \( T_0 \xrightarrow{\text{om}} MBT0 \xrightarrow{MB\beta_{\text{om}}} MBMB1 \), and \( f_2 \) is routinely shown to map \( p_1 \) to \( \phi_1 \) and \( p_2 \) to \( \phi_2 \). Hence \( g_2(d_1) = g_2 \circ \beta_{\text{om}}(p_1) = f_2(d_1) = \phi_1 \), and similarly for \( p_2 \). We have \( \phi_1 \neq \phi_2 \), which implies that \( d_1 \neq d_2 \); hence the final coalgebra morphism distinguishes the behaviour of the terms \( p_1 \) and \( p_2 \).

More generally, a similar phenomenon occurs for \( n \)-step behaviours; every occurrence of \( M \) in the objects \((MB)^n1\) indicates a complete description of how each transition step depends on the store, and modifies it. This is the source of the fine-grained information in the final coalgebra; to obtain a more coarse-grained semantic domain, we aim to record only the relationship between the initial and final stores, and not how the store is modified at every execution step. Formally, this means removing all but the first occurrence of \( M \), and replacing each object \((MB)^n1\) with \( MB^n1\).

On a formal level, a very similar situation was discussed in Section 2.2.3, in the context of non-deterministic lts’s. One obtains a coarse-grained, trace-like equivalence by considering almost the same sequence of characterisations of \( n \)-step behaviour, shown below.

\[ 1 \leftarrow MB0 \leftarrow MBB0 \leftarrow MBBB0 \leftarrow \cdots \]

In the context of stateful programs, these objects serve as a more appropriate description of \( n \)-step behaviour. To illustrate, the 2-step behaviours of the programs \( p_1 \) and \( p_2 \) can be characterised by the following function, in \((MBB0)_P = ((V_P + (V_P + 0)) \times S)^S\), where we only record the relationship between the initial store \( s \), and the final store \( s[x \mapsto 1] \) after execution.

\[ \phi'_1 : \lambda s. (\text{inr}(\text{inl}(\ast)), s[x \mapsto 1]) \]

Under the assumptions of Theorem 2.2.16, the limit of the previous diagram (in \( \text{Kl}(M) \)) describes a final Kleisli coalgebra, formally given by \( M\overline{D} \), where \( \overline{D} \) is the initial \( B \)-algebra. We have glossed over the point that the left-strictness assumption would imply \( M0 = 1 \), which is not the case for the side-effect monad! This will not be a problem in the ordered setting of
3.1. Applying MOS To Programming Languages Directly

\( \mathbb{Cpo}_{\perp} \) which we adopt in Section 3.3.2, but in Set, one could introduce explicit divergence to the monad, \( MX = (1 + (S \times X))^S \), as in [ASP11].

To interpret the resulting semantic domain \( M\overline{D} \), one may consider \( \overline{D} \) to describe completed execution traces, and \( M\overline{D} \) collections of traces; the final coalgebra morphism \( \beta_{\text{om}} : T0 \rightarrow \overline{D} \), of underlying type \( T0 \rightarrow M\overline{D} \), assigns to each program term in \( T0 \) the collection of traces it exhibits.

Instantiated in the context of \( \text{While} \), the initial \( (V + X) \)-algebra is \( \overline{D} = \mathbb{N} \cdot V \): an \( \mathbb{N} \)-fold copower, or coproduct, of \( V \), whose elements we write as \( (n, v) \). They may be thought of as execution traces, described by the number of steps-to-termination \( n \) and the return value \( v \). An element of the object \( M\overline{D} = (\overline{D} \times S)^S \) is then a collection of traces \( (n, v) \), one for each initial store \( s \in S \), additionally decorated with the corresponding final store \( s' \); the intermediate store-manipulations are ignored. This suggests that we may reach a more appropriate characterisation of behavioural equivalence for programs by taking the semantic domain to be a final Kleisli-coalgebra. The following section explores the implications of this decision.

Remark 3.1.11. We conclude this section by commenting on the existence of the final \( MB \)-coalgebra. In the category \( \mathbb{Cpo}_{\perp} \) which we adopt for our later examples, it will be enough to show that \( MB \) is locally continuous; in the case of Set, it is not so straightforward. Much research has focused on final coalgebras for finitary (\( \aleph_0 \)-accessible) functors; however, the side-effect monad, and most of the monads we consider later, are not finitary, due to the presence of countable products. Rather, they are \( \aleph_1 \)-accessible. One may modify existing results to handle \( \aleph_1 \)-accessibility; another approach is to exploit existing results concerning accessible functors on locally presentable categories, which are guaranteed to have final coalgebras (Corollary 3.13 of [PW98] and Corollary 20 of [Wor99]). Set is locally presentable, so it is sufficient to show that \( MB \) is accessible for a final coalgebra to exist. This automatically implies that it is the limit of the final sequence.

In Set, (as for other regular cardinals) \( \aleph_1 \)-accessibility is equivalent to the property that \( M \) coincides with the functor \( M'X = \bigcup_{n<\aleph_1} \bigcup_{f \in X^n} Mf[Mn] \) ([AGT10] Section 3). This property may be checked directly for the side-effect monad, and also for the functor \( BX = V + X \); hence, it also holds for the composition \( MB \).

In the case that \( M \) is a free \( \Delta \)-algebra functor for some \( \Delta \) (as considered later), one may instead use the above method to check that \( \Delta \) is \( \aleph_1 \)-accessible; this may be used to show that \( M \) is too, via an application of the results in [Kel80].
Chapter 3. MOS in a Mixed Kleisli Category for Syntactic Effects

3.2 Introducing a Mixed Kleisli Setting

The framework of Turi and Plotkin must be adapted to accommodate a semantic domain given by a final coalgebra \((\mathcal{D}, \sigma)\) in a Kleisli category. One might hope to achieve this simply by taking the base category \(C\) to be a Kleisli category \(\text{Kl}(M)\). The results of Turi and Plotkin would carry through, and if one could formally define an abstract OS specification, one would obtain an operational model \(\text{om} : T0 \to \mathcal{B}T0\), and the final coalgebra morphism \(T0 \to \mathcal{D}\), of underlying type \(T0 \to M\mathcal{D}\), would give the characterisation of behaviour sought in the previous section; moreover, the semantics induced by the assignment \(\overline{\beta}_{\text{om}}\) would be automatically adequate and compositional.

However, this is not possible in general because it is difficult to define functorial syntax in a Kleisli category. Previously, we assumed the base category \(C\) had products and coproducts, which allowed us to build polynomial syntax functors. The Kleisli category inherits coproducts from the base category \(C\), as the inclusion functor \(J : C \to \text{Kl}(M)\) (Definition 2.2.3) is a left adjoint, and hence preserves colimits; however, there is no guarantee that products exist in \(\text{Kl}(M)\), and there is no obvious workaround. Hence, we have no generic way of defining syntax functors.

Remark 3.2.1. An important exception to this is when the monad \(M\) is commutative. Typical examples of commutative monads \(M\) include the power-set, sub-distribution and bag monads. These are more frequently associated with the semantics of process algebras, rather than programming languages; the computational monads occurring in program semantics, such as the side-effect monad, are typically not commutative. For this reason, we focus on the non-commutative case.

One may explain the problem by arguing that syntax does not naturally belong in the Kleisli category. Kleisli arrows \(X \to MY\) are concerned with the propagation of effects, which is natural in the context of program behaviour; whereas arrows \(\Sigma f : \Sigma X \to \Sigma Y\) between syntax constructors serve the purpose of relabelling arguments. The link between syntax and behaviour is given by the final \(\mathcal{B}\)-coalgebra map \(\overline{\beta}_{\text{om}} : T0 \to M\mathcal{D}\), which may be seen as an arrow in both categories. This suggests the possibility of ‘syntax in the underlying category, and behaviour in the Kleisli category’; this amounts to replacing Diagram 2.1, shown again on the left below, with the diagram on the right. (We call the new denotational model \(\overline{\text{dm}}\), induced in a different way, to distinguish it from the model \(\text{dm}\) of the original framework; however, we will still induce an operational model \(\text{om}\) using the basic method of structural recursion.)
Three main requirements arise in attempting such an adaptation:

- We must ensure we can construct the semantic domain \( \overline{M\mathcal{D}} \), so that we can obtain canonical maps \( \overline{\beta} \gamma \) from coalgebras \((X, \gamma)\) into \( \overline{M\mathcal{D}} \) giving a suitable characterisation of program behaviour.

- We must demonstrate the feasibility of defining abstract OS specifications, so that we can induce operational and denotational models \( \text{om}, \overline{\text{om}} \).

- We must ensure the resulting denotational semantics \([-]\) is compositional and adequate with respect to the behavioural equivalence induced by the map \( \overline{\beta} \text{om} \).

After introducing the necessary definitions, the rest of this chapter tackles each of the above requirements in turn. However, it is hard to make progress on these goals without a more detailed inspection of the monads involved. This suggests that we consider languages in terms of computational effects, rather than monads; but the machinery of Lawvere theories is rather involved for this task. For that reason, we make a preliminary analysis of languages whose executions introduce purely syntactic effects, with no equations assumed on the effects. The operational models for such languages form a class of transition systems, which we will call syntactic effectful transition systems (ETS’s).

### 3.2.1 Effect Syntax and Behaviour

#### Effect Syntax

To construct both programs and effect syntax, we assume we are given two syntactic signatures: an \( \mathcal{S} \)-signature \( \text{Sig} \) for program syntax constructors, and, for simplicity, a single-sorted signature \( \text{Eff} \) for effects, which are allowed to occur at every sort in \( \mathcal{S} \). (Countable arities are required for operators like \( \text{rd}_x \).)
Example 3.2.2. The single \((s)\)-sorted signature \(\text{Eff}\) for global store (with locations \(L\)) consists of \(\mathbb{N}\)-ary operators \(\text{rd}_x : s^\mathbb{N} \to s\) for each location \(x \in L\), and a unary operation \(\text{wr}_{x,n} : s \to s\) for each \(x \in L, n \in \mathbb{N}\); and the signature for non-determinism contains a single binary operator \(\text{or} : s^2 \to s\).

In analogy to the functor \(\Sigma\) constructing multi-sorted syntax, we may define an effect-syntax functor \(\Delta\). In a base category \(C\) with coproducts and countable products, a single-sorted signature \(\text{Eff}\) gives rise to a polynomial functor \(\Delta_0 : C \to C\), as did the syntax constructors in the previous chapter; applying this functor at each sort \(s \in S\) gives an effect-syntax functor \(\Delta : C^S \to C^S\), with \((\Delta X)_s = \Delta_0 X_s\).

Just as the free \(\Sigma\)-algebra monad \(T\) allows us to build program syntax \(TX\) over syntax variables \(X\), we can use the free \(\Delta\)-algebra monad, which we will call \(Te\), to build effect-syntax terms over \(X\). It is given by applying the free \(\Delta_0\)-algebra monad, which we call \(Te_0\), identically at each sort. An example of such a term is \(\text{wr}_l,3(\text{wr}_l,4(x_1; x_2))\) in \((Te)X)\), where \(x_i\) are program variables of command sort (i.e. elements of \(X_P\)). We write such terms as \(\delta((x_i)_{i \in I})\), where \(\delta\) is an arbitrary effect-syntax tree with \(I\)-indexed leaves given by \(x_i\). It is important to note that these effect-trees may be trivial, containing no effect syntax; this gives rise to ‘singleton’ terms \(x\), or more formally \(\eta_{Te_X}^X(x)\), which arise by applying the unit \(\eta_{Te_X}^X : X \to Te_X\) of the monad \(Te\) to \(X\). (For convenience, we sometimes omit the superscript and write \(\eta_X\).)

For convenience, we make frequent use of the following notation: we often write \(\tilde{x}\) instead of \((x_i)_{i \in I}\), for an \(I\)-indexed collection of arguments \(x_i\), so that we may write syntax terms as \(\sigma(\tilde{x})\) instead of \(\sigma((x_i)_{i \in I})\).

### Effectful Behaviour

In analogy to the transitions of \textbf{SWhile} from Section 3.1.1, we assume that evaluating an \(s\)-sorted term \(x\) proceeds in atomic, unlabelled transition steps \(x \to x'\), and may terminate with a return value \(x \to q\) drawn from \(V_s\). We also assume that program execution may introduce a syntax-tree \(\delta\) of effects given by the signature \(\text{Eff}\); so that instead of one atomic transition, a program may exhibit an effect-syntax tree of transitions, as illustrated by the following example.

In general, this is a different sort of operational model from the ones we have considered already (e.g. \textbf{While}) which involved computational monads; we will call it a \textit{syntactic effectful transition system}, or syntactic ETS.

Example 3.2.3. We contrast transitions of the standard operational model for \textbf{While}, and its representation in terms of syntactic effects, which we will call \textbf{sEWhile} (‘syntactic effectful \textbf{While}’), defined in Section 3.3.4. As shown below, consider the program \(x = 5; x = 1\), where \(x\) is a store-variable; to evaluate this in terms of program-store pairs \((p, c)\) in the standard...
3.3 A Semantic Domain for Syntactic Effects, through Order-Enrichment

operational model, we modify $x$ in the store at each step, as shown. In an effectful operational model, we abstract away the implementation details of the store, and simply record whenever an update is required, by introducing unary effect-symbols $\text{wr}_{x,5}$ and $\text{wr}_{x,1}$, as shown. (On a formal level, these two-step transition behaviours, accumulating state-updates and effects respectively, are represented by coalgebraic iteration in the Kleisli categories for the side-effect monad and the effect-syntax monad.)

$$\text{While: } \langle x = 5; x = 1, c \rangle \to \langle x = 1, c[x \mapsto 5] \rangle \to \langle \ast, c[x \mapsto 1] \rangle$$

$$\text{sEWhile: } x = 5; x = 1 \to \text{wr}_{x,5}(x = 1) \to \text{wr}_{x,5}(\text{wr}_{x,1}(\ast))$$

By contrast, to evaluate the expression $x + 5$ in While, one looks up the value $c(x)$ of $x$ in the store and evaluates $c(x) + 5$ as shown; whereas in an effectful operational model, one introduces an $\mathbb{N}$-ary effect-symbol $rd_x$ to record the fact that the computation of $x + 5$ ‘branches’ into multiple execution paths $(n + 5)_{n \in \mathbb{N}}$, each of which has to be evaluated independently.

$$\text{While: } \langle x + 5, c \rangle \to \langle c(x) + 5, c \rangle \to \langle +_{c(x)}(5), c \rangle \to \langle c(x) + 5, c \rangle$$

$$\text{sEWhile: } x + 5 \to rd_x((n + 5)_{n \in \mathbb{N}}) \to rd_x((+_{n}(5))_{n \in \mathbb{N}}) \to rd_x((n + 5)_{n \in \mathbb{N}})$$

To represent a syntactic ETS as a coalgebra, we note that in general, each state $x$ of the state-space $X$ is assigned an effect-syntax tree $\delta((b_i)_{i \in I})$ (which we also write $\delta(\tilde{b})$) of atomic transitions $b_i$ which describe either a new state $x'_i$ or a value $v'_i$. These are elements of $BX = V + X$; the syntactic effect-trees over such transitions $BX$ are given by $T_e BX$. Thus a syntactic ETS assigns to each state in $X$ an element of $T_e BX$, making it a $T_e B$-coalgebra.

3.3 A Semantic Domain for Syntactic Effects, through Order-Enrichment

The syntactic ETS fits into the scheme of ‘operational models as $MB$-coalgebras’ we have introduced, where $BX = V + X$ for some collection of values given by $V$, and the monad $M$ is taken to be the free effect-syntax functor $T_e$, given componentwise by the monad $T_{e_0}$ for the effects of some signature Eff. In analogy to Section 3.1.3 where we considered semantic domains for While programs, here we show that the final $T_e B$-coalgebra $D$ is again a very fine-grained semantic domain, and that the candidate domain of $M \overline{D} = T_e \overline{D}$ is again a more appropriate choice. We then explore the order structure required to ensure a final $T_e B$-coalgebra exists, and show how these requirements are met by adopting a base category of $\text{Cpo}_{\leq \dagger}$. 
3.3.1 The Final $T_eB$-coalgebra in the Context of Syntactic Effects

As before, the final $T_eB$-coalgebra is approximated by the objects $(T_eB)^n$ of the final sequence, which characterise the possible $n$-step behaviours of states $x$. In $\text{Set}$, the first approximant $T_eB1$ is the set of finite-depth effect-syntax trees $d = \delta(\tilde{b})$ of atomic behaviours, where any successor states are represented by the element $\text{inr}(\ast)$ of $B1 = V + 1$. An example is the behaviour

$$\text{rd}_x(\text{inr}(\ast), \text{inl}(5), \text{inr}(\ast), \text{inl}(5), \ldots)$$

– describing a lookup on $x$, followed by immediate termination with return value 5 if $x$ is odd; otherwise, a transition to other (unknown) states $\ast$. We may represent this behaviour by a syntax-tree as follows, omitting the inclusions $\text{inl, inr}$:

In general, multiple effects may occur, giving rise to a syntactic effect-tree $\delta$. Rotating the above tree 90 degrees anticlockwise (and labelling the tree $d$), we may represent the form of these 1-step behaviours by a simple schematic diagram

$$d \xrightarrow{\delta} \text{v'}$$

where each effect-tree $\delta$ is reduced to a triangle; the leaves are either return values $v$ or the representation $\ast$ of an arbitrary state, and these are collected into two cases as shown. They are decorated $'$ to indicate they occur at the first transition. This schematic notation will be used in our syntactic proof of adequacy and compositionality, in Section 3.4.4.

We now consider the second object $T_eB(T_eB1)$ of the final sequence, which describes 2-step transition behaviour. It contains effect-syntax terms $d$ whose arguments are either: values $\text{inl}(v)$, i.e. computation branches which immediately terminate with return value $v$; or (injections of) elements $\text{inr}(d')$ of the first object $T_eB1$ that we just considered, corresponding to the 1-step behaviours of any successor states. An example of such an element $d$ is

$$\text{wr}_{x,3}(\text{inr}(\text{wr}_{y,4}(\text{inl}(\ast))))$$

corresponding to the transition behaviour of the term $x = 3; y = 4$, updating $x$ and then $y$ in two steps. Note that this is a distinct element from the behaviour

$$\text{wr}_{x,3}(\text{wr}_{y,4}(\text{inr}(\text{inl}(\ast))))$$
which indicates that both $x$ and $y$ are updated in the first transition step. Once again, this is undesirably fine-grained information contained in the final $T_eB$-coalgebra, distinguishing behaviours that are observably identical.

In the same way as before, such 2-step behaviours may be represented by a schematic diagram as shown. The left-most triangle represents the effects occurring at the first execution step of the behaviour, given either by values $v'$ or non-terminal transitions, whose ensuing execution steps are described by the leaves $d'$; these are 1-step behaviours as described above.

\[
\begin{array}{c}
  d \\
  \delta \\
  \downarrow v'
\end{array}
\quad
\begin{array}{c}
  d' \\
  \delta' \\
  \downarrow v''
\end{array}
\quad
\begin{array}{c}
  d'' \\
  \delta'' \\
  \downarrow v'''
\end{array}
\quad
\begin{array}{c}
  \ddots
\end{array}
\]

By considering the approximants to the final coalgebra, the method of Section 3.1.3 and Remark 3.1.10 allow us to show that the final $T_eB$-coalgebra morphism $\beta_{om} : T0 \rightarrow D$ distinguishes programs like $(x = 3; x = 4)$ and $(x = 0; x = 4)$ – on this occasion, because they produce different syntactic effects at each execution step, $\text{wr}_3(\text{inr}(\text{wr}_4(\text{inl}(\ddots))))$ and $\text{wr}_0(\text{inr}(\text{wr}_4(\text{inl}(\ddots))))$, rather than different store-manipulations. This again demonstrates the fine-grainedness of final $MB$-coalgebras as semantic domains; in this context, we want to characterise programs by the overall effect-trees they produce, rather than the effects at each execution step.

The above comments apply for the other approximants $(T_eB)^n1$, generalising to $n$ layers of effects $\delta', \delta'', \ldots, \delta^{(n)}$ – where superscript $(r)$ denotes $r$ occurrences of $\cdot$. This leads us to expect that the final $T_eB$-coalgebra, described by such approximants, will consist of possibly-infinite trees of this form; and this may be proven to be the case in $\text{Set}$ and $\text{Cpo} \perp$.

**Remark 3.3.1.** We conclude this section by sketching how this may be proven in $\text{Set}$; the situation in $\text{Cpo} \perp$ will be considered in the next section. The first point is that although the functor $T_eB$ is not finitary, the final $T_eB$-sequence converges in $\omega$ steps because it is syntactic in nature. To prove this, it is enough to show that the functor $T_eB$-preserves the limit $(T_eB)^{\omega^p}1$ of the final $T_eB$-sequence up to $\omega$. Generally in $\text{Set}$, the limit of an $\omega^p$ chain $X_1 \overset{f_1}{\leftarrow} X_2 \overset{f_2}{\leftarrow} X_3 \overset{f_3}{\leftarrow} \cdots$ is the set of sequences $(x_1, x_2, x_3, \ldots)$ such that $x_r = f_r(x_{r+1})$. In the case of the final sequence up to $\omega$, where $X_r = (T_eB)^r1$, each sequence in the limit corresponds to an infinite behaviour

\[
\begin{array}{c}
  d \\
  \delta \\
  \downarrow v'
\end{array}
\quad
\begin{array}{c}
  d' \\
  \delta' \\
  \downarrow v''
\end{array}
\quad
\begin{array}{c}
  d'' \\
  \delta'' \\
  \downarrow v'''
\end{array}
\quad
\begin{array}{c}
  \ddots
\end{array}
\]

where the $r^{th}$ term $x_r$ of the sequence is given by truncating the behaviour at the $r^{th}$ level (corresponding to $\delta^{(r)}$ in the above diagram), and replacing the omitted sub-trees $d^{(r)}$ with leaves $\ast^{(r)}$. Applying $T_eB$ to the limit (and the limiting cone) gives an essentially equivalent
set, \( T_eB(T_eB \omega 1) \), containing one layer \( \delta \) of effects, with leaves given either by values \( v' \) or such sequences of \( r \)-depth approximants \( (x_1, x_2, \ldots) \). It is straightforward to show there is an isomorphism \( T_eB(T_eB \omega 1) \cong T_eB \omega 1 \) which is a morphism of cones; this implies that \( T_eB \) preserves the limit, as required.

### 3.3.2 The Semantic Domain in a Kleisli category

We now outline the structure of the semantic domain for effectful programs given by a final coalgebra in the Kleisli category for the monad \( T_e \). Given a lifting \( \overline{B} \) of the functor \( BX = V + X \), if we aim to apply the limit-colimit results of Section 2.2.3, then the final \( \overline{B} \)-coalgebra in \( Kl(T_e) \) is given by the initial \( \overline{B} \)-algebra \( \overline{D} \); and the final-coalgebra morphisms \( X \to \overline{D} \) in the Kleisli category are of underlying type \( X \to T_e \overline{D} \), which makes \( T_e \overline{D} \) into a semantic domain.

We begin by considering this semantic domain in the context of \( Set \). The initial \( B \)-algebra is given by \( \overline{D} = N \cdot V \), consisting of pairs \( (n, v) \) which we may interpret as a number of steps-to-termination \( n \), and a terminal value \( v \) (at any sort). Then \( T_e \overline{D} \) consists of syntactic effect-trees, with leaves given by these pairs \( (n, v) \); the idea is that during execution, a program accumulates an effect-tree of execution paths, and each path may terminate after some number of steps \( n \) with some value \( v \); the pairs \( (n, v) \) characterise these execution paths. As a simple example, the following effect-tree describes the behaviour of a program which looks up the value \( n \) of variable \( x \), and returns 42 after \( n \) transitions:

\[
\text{rd}_x((0, 42), (1, 42), (2, 42), \ldots).
\]

However, this semantic domain does not account for non-terminating programs. The intention is for each program to be assigned an effect-tree whose leaves characterise its execution paths; thus, in addition to pairs \( (n, v) \) describing terminating computation paths, we aim to represent divergent computation paths by a value \( \bot \). In the same way as before, we may display the form of such behaviours diagrammatically:

\[
\begin{array}{c}
\bot \\
(1, v') \\
(2, v'') \\
\vdots
\end{array}
\]

On a syntactic level, one may simply introduce the constant \( \bot \) to the syntax signature \( \text{Eff} \) to represent divergence, as anticipated by adding an extra element to the side-effect monad \( MX = (1 + (X \times S))^S \) in \( Set \). However, shortly we show how this effect may be achieved by working in the ordered setting of \( Cpo_{\bot} \).
3.3. A Semantic Domain for Syntactic Effects, through Order-Enrichment

We now review, in increasing order of difficulty, the conditions required to apply Theorem 2.2.16 and ensure we can take $T_e \overline{D}$ as a semantic domain, where $BX = V + X$:

1. The functor $BX = V + X$ has an initial algebra $\overline{D} = N \cdot V$, and a lifting $\overline{B}$ into $\text{Kl}(M)$.

2. The Kleisli category $\text{Kl}(M)$ is $\text{Cpo}_\perp$-enriched, and $\overline{B}$ is locally continuous.

3. Composition in $\text{Kl}(M)$ is continuous and left-strict.

1. **An initial algebra and lifting of $B$.** This requirement is not a problem; an initial algebra for the simple functor $BX = V + X$ is easy to come by, both in $\text{Set}$ (by Remark 2.1.8) and in $\text{Cpo}_\perp$ (by Theorem 3.3.7 below). In addition, we recall (Example 2.2.10) that one may define a distributive law of $B$ over any monad $M$, which by Lemma 2.2.8 defines a lifting $\overline{B}$.

2. **Order-enrichment.** This requirement poses more difficulties.\(^1\) We require an order-relation $\sqsubseteq$ on Kleisli-arrows $f', g' : Y \to Z$, of underlying type $f, g : Y \to T_e Z$, that will make them into a cppo.

This suggests to equip $T_e Z$ with a cppo-structure $\sqsubseteq_Z$, for any $Z$; then we will have a pointwise order on Kleisli-arrows, namely $f \sqsubseteq g$ iff $f(y) \sqsubseteq_Z g(y)$ for all $y$ in $Y$, and this will make the arrows into a cppo as required.

The role of this order structure is illustrated by Proposition 2.2.18 and Example 2.2.19: the behaviour of a $\overline{B}$-coalgebra state $x$ is given by the join of a sequence of approximants $\beta^{(n)}(x)$ which essentially unravel the coalgebraic behaviour of $x$ for $n$ steps, before replacing any incomplete executions by $\perp$. The result is a sequence of effect-trees, each of which is obtained by the previous one by replacing leaves containing ‘incomplete executions’ $\perp$ with the next step of behaviour of that execution path. This construction will be described in more detail in Example 3.3.12; for instance, a program which may non-deterministically return $\perp$ after $n$ transitions would give the following sequence of approximations:

\[
\text{⊥ or (1, 1) or (2, 2) or } \perp \subseteq (1, 1) \text{ or ((2, 2) or } \perp) \subseteq (1, 1) \text{ or ((2, 2) or } ((3, 3) \text{ or } \perp)) \subseteq \cdots
\]

The natural order-structure implied by this sequence is as follows: incomplete executions $\perp$ should be considered below all other behaviours, so that we can take the join of successive approximants $\beta^{(n)}(x)$ as they unravel the behaviour of states $x$. In the context of arbitrary effect-trees, we thus take $\delta(\overline{y}) \sqsubseteq \epsilon(\overline{z})$ iff the latter can be obtained by ‘extending’ the undefined leaves of the former, by replacing some occurrences of $\perp$ with new effect-trees.

---

\(^1\)A trivial solution would be to just add a bottom element $\perp$ to $T_e$, and make it into a flat cppo. However, this gives an unsatisfactory treatment of divergence: one may check (using the approximants $\beta^{(n)}$ of Proposition 2.2.18) that the resulting coalgebra morphisms $X \to T_e(N \cdot V)$ assign each coalgebra-state $x$ to the bottom element, unless every execution path of $x$ terminates.
This gives us a pointed partial order structure on terms. However, it is not $\omega$-complete; there is no join of the chain shown above. This means we must allow infinitely deep effect-syntax terms. As we argued in [ASP11], this may be achieved in Set by replacing the free $\Delta$-algebra monad $T_e$ with a functor $F_{\Delta}$, which maps each $X$ to the final $(X + \Delta)$-coalgebra. This is again a monad, as shown in [GLDMP01].

A more elegant approach, described in the next section, is to move to the base category $\mathsf{Cpo}_{\perp!}$ of ccppos and strict $\omega$-continuous functions. Then each object $Z$ carries a cppo-structure $\sqsubseteq_Z$.

By defining syntax functors $\Delta$ in this setting, we will implicitly define an order-structure on terms $T_e Z$; and the arrows $Y \to T_e Z$ thus have a pointwise cppo-structure. On a concrete level, moving to $\mathsf{Cpo}_{\perp!}$ requires extending the order-structure on effect-trees: $\delta((z_i)_{i \in I}) \sqsubseteq \epsilon((z_j)_{j \in J})$ (for $I \subseteq J$) iff the latter can be obtained by extending the former as above, and/or by replacing some $z_i$ with $z'_i$ such that $z_i \sqsubseteq z'_i$.

Lastly, local continuity of $B$ is then a straightforward consequence of the local continuity of $B$, in the setting of $\mathsf{Cpo}_{\perp!}$.

3. Continuity and Left-strict composition. To address this requirement, we begin by considering composition of Kleisli-arrows concretely: in $\mathsf{Kl}(T_e)$, the composition of arrows $f' : X \to Y$ and $g' : Y \to Z$ corresponds to the underlying composition

$$X \xrightarrow{f} T_e Y \xrightarrow{T_e g} T_e^2 Z \xrightarrow{\mu_Z} T_e Z.$$  

The interpretation is as follows: the first arrow maps each $x$ in $X$ to an effect-tree $f(x) = \delta((y_i)_{i \in I})$ over $Y$; the second arrow maps each leaf $y_i$ of this tree to another effect-tree $\epsilon_i(\tilde{z}_i)$ over $Z$, giving nested trees $\delta((\epsilon_i(\tilde{z}_i))_{i \in I})$; and finally, the multiplication ignores the formal distinction between the two layers of effect-syntax.

Continuity of Kleisli-composition can be handled as follows. In $\mathsf{Set}$, we would have to prove continuity directly, by a rather involved argument; but more conveniently in $\mathsf{Cpo}_{\perp!}$, continuity of composition carries over into the Kleisli category $\mathsf{Kl}(M)$, as follows.

$$g' \circ \sqcup_{n<\omega}(f_n) = \mu_Z \circ T_e g \circ \sqcup_{n<\omega}(f_n) = \sqcup_{n<\omega}(\mu_Z \circ T_e g \circ f_n) = \sqcup_{n<\omega}(g' \circ f_n)$$

Left-strictness is the most important of all the requirements: “post-composing with the bottom arrow $\bot$ gives $\bot$”. Recall that this means that for all arrows $f' : X \to Y$, we have $\bot \circ f' = \bot$ (in $\mathsf{Kl}(T_e)$). The bottom arrow $\bot_{Y,Z} : Y \to Z$ is given by the underlying bottom arrow $\sqcup_{Y,T_e Z} : Y \to T_e Z$ in $\mathsf{Cpo}_{\perp!}$, mapping every $y$ to $\bot$.

2As an alternative, the non-strict category $\mathsf{Cpo}_{\perp}$ is not suitable as it does not have categorical coproducts, which makes it difficult to reason about functorial syntax, such as “the initial $(X + \Delta)$-algebra".
Now we consider what it means to post-compose an arrow \( f : X \to T_e Y \) with the bottom arrow \( \bot_{Y,Z} \) in the Kleisli category. First, for each \( x \) in \( X \), the function \( f \) produces an effect-tree \( \delta((y_i)_{i \in I}) \) with leaves in \( Y \) (including \( \bot \)); then the leaves \( y_i \) are uniformly sent to \( \bot \), resulting in the effect-tree \( \delta((\bot)_{i \in I}) \). For this action to coincide with the bottom arrow \( \bot'_{X,Z} = \bot_{X,T_e Z} \), as required for left strictness, we must identify the effect-tree \( \delta((\bot)_{i \in I}) \) with \( \bot \): any effect-tree whose leaves are all divergent \( \bot \) is indistinguishable from divergence \( \bot \). Similarly, any nullary effects (like exceptions) must also be identified with divergence \( \bot \).

We may rephrase this constraint as follows: ‘replacing all leaves with \( \bot \) is the same as replacing the whole tree with \( \bot \).’ On a more abstract level, this suggests that the functor \( T_e \) should satisfy \( T_e \bot_{Y,T_e Y} = \bot_{Y,T_e Y} \); in other words, it is strict (\( \bot \)-preserving). In practice, we will guarantee this by showing it is a \( \text{Cpo}_{\bot!} \)-functor:

**Definition 3.3.2.** An endofunctor \( F \) on a \( \text{Cpo}_{\bot!} \)-category \( \mathcal{C} \) is \( \text{Cpo}_{\bot!} \)-enriched – equivalently, a \( \text{Cpo}_{\bot!} \)-functor – if it is locally continuous and strict, in the sense that it preserves bottom arrows: \( \bot_{X,Y} = \bot_{FX,FY} \).

It is easy to check directly that strictness of \( T_e \) is enough to guarantee left-strictness of Kleisli composition: \( X \xrightarrow{g'} Y \xrightarrow{T_e} Z \) corresponds in \( \text{Cpo}_{\bot!}^S \) to the following composition. (Recall that the bottom Kleisli-arrow \( \bot'_{Y,Z} \) is given by \( \bot_{Y,T_e Z} \).

\[
\begin{align*}
X \xrightarrow{g} T_e Y \xrightarrow{T_e \bot_{Y,T_e Z}} T^2_e Z \xrightarrow{\mu_Z} T_e Z
\end{align*}
\]

Finally, strictness of composition in \( \text{Cpo}_{\bot!}^S \) implies that pre- and post-composing the middle arrow \( \bot_{T_e Y,T_e Z} \) with the others gives \( \bot_{X,T_e Z} = \bot'_{X,Z} \) as required. This implies that if a monad \( M \) on a \( \text{Cpo}_{\bot!} \)-category \( \mathcal{C} \) is strict, then the Kleisli-category \( \text{Kl}(M) \) has left-strict composition.

We now summarise the implications of this section in the context of \( \text{Cpo}_{\bot!}^S \), as follows:

**Corollary 3.3.3.** Let \( M \) be a locally continuous and strict monad – equivalently, a \( \text{Cpo}_{\bot!}^S \)-monad – on the category \( \text{Cpo}_{\bot!}^S \). Then the functor \( BX = V + X \) has a lifting \( \bar{B} \) to \( \text{Kl}(M) \) given by the distributive law of equation 2.2.9; and the final \( \bar{B} \)-coalgebra in \( \text{Kl}(M) \) is given by \( (D, \alpha^{-1} : D \to \bar{B}D) \) where \( (D, \alpha) \) is the initial \( B \)-algebra. Moreover, let \( \bar{\beta}_\gamma : X \to M\bar{D} \) be the underlying arrow of the \( \bar{B} \)-coalgebra morphism from \( (X, \gamma) \) into the final \( \bar{B} \)-coalgebra. Then \( \bar{\beta}_\gamma \) is given by the join of the arrows \( \bar{\beta}_\gamma^{(n)} : X \to M\bar{D} \) of Proposition 2.2.18.

### 3.3.3 The Base Category of \( \text{Cpo}_{\bot!} \)

We conclude Section 3.3 by showing how functorial syntax in \( \text{Cpo}_{\bot!} \) satisfies the three requirements listed above and ensures existence of a final Kleisli-coalgebra, as a semantic domain for the syntactic ETS.
The first point is that $\mathbf{Cpo}_\bot!$ has the required structure, illustrated below, to define polynomial syntax. The categorical product of two cppos $X, Y$ is given by the cartesian product $X \times Y$, which consists of pairs $(x, y)$ under the componentwise order; the pair $(\bot, \bot)$ is the bottom element (also written $\bot$). There is also the smash product $X \otimes Y$, which in addition identifies the bottom element with the pairs $(x, \bot)$ and $(\bot, y)$. The coproduct is the coalesced sum $X \oplus Y$, given by taking the disjoint union of $X$ and $Y$, and identifying the bottom elements; and there is the ‘lifting’ construction $(\dashv)\bot$ which attaches a new bottom element $\bot'$ to a given cppo. The two products are related as follows: $X \times Y \simeq (X \otimes Y) \bot$. Lastly, the initial and final objects are both given by the one-element cppo, which we write $0$.

**Example 3.3.4.** Suppose $X$ and $Y$ are two-element flat cppos, with elements $\{\bot, x\}$ and $\{\bot, y\}$ respectively. The above constructions are given by:

$X \oplus Y : \begin{array}{ccc} x & \downarrow \bot \end{array}$ \quad $X \times Y : \begin{array}{ccc} (x, y) \downarrow (x, \bot) \downarrow (\bot, y) \end{array}$ \quad $X \otimes Y : \begin{array}{ccc} (x, y) \downarrow \bot \end{array}$ \quad $(X)_{\bot} : \begin{array}{ccc} x \downarrow \bot \end{array}$

We overload notation and write $A_{\bot}$, when $A$ is a set, for the flat cppo obtained by adding a bottom element to $A$. As we have done in the previous section, we will continue to use the symbol $+$ for the coalesced sum $\oplus$ to emphasise that it is the categorical coproduct in $\mathbf{Cpo}_\bot!$.

Another key property of $\mathbf{Cpo}_\bot!$ is that it is symmetric monoidal closed (Definition 2.1.16), with respect to the smash product $\otimes$ (with monoidal unit given by the two-element cppo $1_{\bot}$) and the strict function space $Y^X$ (consisting of the strict and $\omega$-continuous functions $X \to Y$ ordered pointwise). This implies that $\mathbf{Cpo}_\bot!$-enriched monads $M$ have a strength $\text{st}_{X,Y} : X \otimes MY \to M(X \otimes Y)$ with respect to the smash product $\otimes$, of which we will make frequent use.

**Ordered Syntax, Initial Algebras (and Final Coalgebras)**

The above constructions in $\mathbf{Cpo}_\bot!$ allow us to build functors $\Sigma, \Delta$ describing the application of syntax constructors $\sigma(\tilde{x})$ to variables in $X$. However, the order-structure gives us more ways of achieving this than we had in $\mathbf{Set}$; in the previous sections, we naturally represented the application of syntax constructors to variables in $X$ through $I$-indexed tuples $((x_i)_{i \in I})$ in an $I$-fold product $X^I$. By contrast, in $\mathbf{Cpo}_\bot!$, given a cppo of variables $X$, one has a choice of Cartesian or smash products, to build tuples $\sigma(x_1, x_2, \ldots)$ of terms, given by applying $n$-ary syntax constructors $\sigma$ to arguments in $X$ (for $n \leq \omega$). For instance, one might represent a ternary constructor by $\Sigma X = X \times (X \times X)$, $X \otimes (X \times X)$, etc. Analogously, we also have a
choice to represent unary syntax constructors either by \( X_\bot \) or \( X \). A set \( N \) of ‘constants’, or nullary operators, may be represented by the flat cppo \( N_\bot \).

We now consider the difference implied by the choice of Cartesian or smash product in \( \Sigma \). However one constructs an \( n \)-fold product of \( X \) from \( \times \) or \( \otimes \), one essentially obtains a collection of tuples \((x_1, x_2, \ldots)\); the difference is in the way undefined arguments are handled.

Using the smash product \( X \otimes (\cdot) \) implies that if a term \( \sigma(\ldots, \bot, \ldots) \) has argument \( \bot \) in this \( X \)-position, the term will be identified with \( \bot \); e.g. in the tuples \((x_1, x_2, x_3)\) of \( X \otimes (X \otimes X) \), we would identify \((\bot, x_2, x_3) = \bot \). By contrast, if we use a Cartesian product \( X \times (\cdot) \), we allow undefined arguments; as tuples of \( X \times (X \otimes X) \), we would consider \((\bot, x_2, x_3) \) to be distinct from \( \bot \). (However, we \textit{would} identify \((x_1, \bot, x_3) = (x_1, x_2, \bot) = \bot \) due to the smash product.)

This means that we should use Cartesian products \( X^n \) to construct syntactic effects in the functor \( \Delta \), because a computation branch should be able to introduce divergence: we would not want to identify a behaviour \( 42 \) or \( \bot \) with divergence \( \bot \), which would occur if \( \Delta \) was defined using smash products. By contrast, we are only interested in program-syntax terms \( \sigma(p_1, \ldots, p_n) \) where each argument is defined; so it would be ideal to define \( \Sigma \) in terms of smash products, rather than Cartesian products. However, we will see that this will make it difficult to define operational specifications in \( \text{Cpo}_\bot \), in Section 3.3.4; hence we will define \( \Sigma \) in terms of both products.

The choice of \( \times \) or \( \otimes \) also has consequences for the syntax terms described by initial algebras, as illustrated by the initial sequence in the following example: non-strict operators, such as \( (\cdot)_\bot \) and \( \times \), induce infinite-depth syntax terms.

\textbf{Example 3.3.5.} Consider a signature with a single constant \( \bullet \) and a unary operator \( \circ(\cdot) \), which may be considered strict \( (\circ(\bot) = \bot) \) or non-strict. The corresponding syntax functors for these cases are \( \Sigma X = 1_\bot \oplus X \) and \( \Sigma' X = 1_\bot \oplus X_\bot \). The initial \( \Sigma \)-sequence may be represented as follows, containing syntax terms where all arguments are defined.

\[
\begin{array}{c}
\bot \rightarrow \bullet \rightarrow \bullet \\
\bot \rightarrow \circ \bullet \rightarrow \bullet \\
\circ \circ \bullet \rightarrow \cdots
\end{array}
\]

There is essentially no order structure on these terms; this suggests that if the initial \( \Sigma \)-algebra exists, it will contain only the finite-depth syntax terms. By contrast, the initial \( \Sigma' \)-sequence
allows partially defined syntax terms, which form a chain.

This suggests that an initial \( \Sigma' \) algebra will contain, in addition to finite-depth syntax terms, a chain \( \bot \sqsubseteq \circ \bot \sqsubseteq \circ \circ \bot \sqsubseteq \cdots \), which must have a limit \( \circ^\omega \) (by \( \omega \)-completeness of cppo’s) – an infinite-depth syntax term.

Having made a suitable choice of product to describe syntactic terms \( \sigma(x_1, \ldots) \), to apply a range of such constructors \( \sigma \) simultaneously, we combine the products via the coalesced sum \( \oplus \); being the categorical coproduct, this plays the same role as the disjoint union in \( \text{Set} \). In this way, we may define syntax functors \( \Sigma \) and \( \Delta \), and also a behaviour functor \( BX = \text{Vals}_\bot \oplus X \), where \( \text{Vals}_\bot \) is a flat cppo of terminal values, given by a set \( \text{Vals} \).

The generalisation to multiple sorts \( S \) is exactly analogous to Section 3.1: in the setting of \( \text{Cpo}_\bot \), an object \( X \) is a tuple of cppos, each of which represents a collection of variables. Given an \( S \)-sorted signature \( \text{Sig} \), we represent the application of a syntax constructor \( \sigma : (s_i)_{i<\alpha} \to s_\sigma \) by an \( s \)-sorted Cartesian \( \prod_{i<\alpha} X_{s_i} \) or smash product \( \bigotimes_{i<\alpha} X_{s_i} \); one then takes the coproduct of all such products at each sort \( s \) (compare with Example 3.1.5). We consider the empty smash product to be the \( \otimes \)-monoidal unit, the two-element cppo \( 1_\bot \); this allows us to describe syntax constants in this format.

Given a collection of sets \( (\text{Vals})_s \) at each sort, we write \( \text{Vals}_\bot \) for the corresponding tuple of flat cppos, \( (\text{Vals}_\bot)_s = (\text{Vals}_s)_\bot \). As this tuple will arise frequently in context of the behaviour functor \( BX = \text{Vals}_\bot \oplus X \) in \( \text{Cpo}_\bot \), we often use the symbol \( V \) for \( \text{Vals}_\bot \), and interchange \( \oplus \) with +, so that we may write \( BX = V + X \). (As mentioned earlier, we may do this because in \( \text{Cpo}_\bot \), coalesced \( \oplus \) and categorical + sums coincide.)

Recall that we have assumed effects occur identically at each sort, given by a signature \( \text{Eff} \). Thus, as before we use suitable (co)products to define a single-sorted syntax functor \( \Delta_0 : C \to C \) representing the constructors in \( \text{Eff} \), and apply it componentwise to define a syntax functor on \( \text{Cpo}_\bot^S \) given by \( (\Delta X)_s = \Delta_0 X_s \). The free \( \Delta \)-algebra functor \( T_e \) is then given componentwise by the free \( \Delta_0 \)-functor \( T_{eq} \).
Existence of Initial Algebras

Having discussed the construction of syntax functors, we address the question of showing that their initial algebras exist. This is conveniently guaranteed in $\mathbf{Cpo}_{\bot}$ by the property of algebraic $\omega$-compactness [Ada95].

Definition 3.3.6. A category $C$ is algebraically $\omega$-compact if:

1. It has a zero object $0$ (i.e. both initial and final);
2. For every locally continuous functor $F$, the initial and final $F$-sequences up to ordinal $\omega$ have a colimit $L$ and a limit $G$ respectively, which are preserved by $F$ – the ‘least and greatest fix points of $F$’; (i.e. they give an initial $F$-algebra $(L, \alpha)$ and final $F$-coalgebra $(G, \zeta)$, by Remarks 2.1.8 and 2.1.23, where $\alpha, \zeta$ are isomorphisms);
3. The unique arrow between the fixpoints $L \to G$ – given by applying initiality to the $F$-algebra $(G, \zeta^{-1})$, or finality to the $F$-coalgebra $(L, \alpha^{-1})$ – is an isomorphism.

This property holds in $\mathbf{Cpo}_{\bot}$, and the multi-sorted version $\mathbf{Cpo}^S_{\bot}$, by the following theorem, shown in loc.cit. and based on the method of $O$-colimits [SP82].

Theorem 3.3.7. Let $C$ be a $\mathbf{Cpo}_{\bot}$-enriched category with an initial object and colimits of $\omega$-sequences of embeddings. Then $C$ is algebraically $\omega$-compact.

The category $\mathbf{Cpo}_{\bot}$ is $\mathbf{Cpo}_{\bot}$-enriched by Example 2.2.14; and it has the required colimits because $\mathbf{Cpo}_{\bot}$ is cocomplete, having colimits of all small diagrams. (Cocompleteness follows from the fact that $\mathbf{Cpo}_{\bot}$ may be shown to be essentially algebraic; this is an easy adaptation of [AR94] p.163. This fact also implies locally countable presentability, which is required in the next chapter.) Lastly, this generalises to multiple sorts $\mathbf{Cpo}^S_{\bot}$, as the colimit of a diagram is given componentwise by colimits of the components of the diagram at each sort.

Thus, every locally continuous functor on $\mathbf{Cpo}^S_{\bot}$ has both an initial algebra and a final coalgebra. It is straightforward to check that all the operations $\oplus, \times, \otimes$ used to define syntax functors $\Sigma, \Delta$ are locally continuous; as before, this gives rise to initial algebras $T_0, T_e0$ respectively, and free algebra functors $T_\Sigma$ (or just $T$) and $T_e$, which may be used to build syntax terms over variables $X$.

Remark 3.3.8. One may seek a concrete characterisation of the initial $\Sigma$-algebra $T_0$ to confirm that it corresponds to the syntax terms of the language as we have described, with the order structure given by Cartesian and smash products in $\Sigma$ described above; and similarly for the terms $TX$ over a cppo of variables $X$. One strategy is the “guess and check” method: one guesses a concrete structure for $T0$ (from the initial $\Sigma$-sequence), and shows it satisfies a
Chapter 3. MOS in a Mixed Kleisli Category for Syntactic Effects

particular property which is uniquely satisfied by the initial $\Sigma$-algebra. A result in this direction, building again on [SP82], is the following, adapted from Theorem 48 of [FM91]. We do not define all the conditions of this theorem, except to say that they are satisfied by $\text{Cpo}_{\perp!}$, as shown in [SP82]. (This generalises to powers $\text{Cpo}_{\perp!}^S$, by generalising embedding-projection pairs and colimits componentwise.)

**Theorem 3.3.9.** Let $\mathcal{C}$ be a localised $O_{\perp!}$-category, such that the subcategory $\mathcal{C}_E$ of embeddings has colimits of $\omega$-chains, and let $F$ be a locally continuous functor on the subcategory $\mathcal{C}_{\perp}$ of objects and $\perp$-preserving morphisms. Then there is an initial $F$-algebra $(I, \alpha)$, characterised uniquely (up to isomorphism) by the following properties:

1. $\alpha$ is an isomorphism;

2. $\sqcup_{n<\omega} \Phi^n(\perp_{I,I}) = \text{id}$, where $\Phi : (I \xrightarrow{\varphi} I) \mapsto (I \xrightarrow{\alpha^{-1}} FI \xrightarrow{F\varphi} FI \xrightarrow{\alpha} I)$.

(Note the similarity between the operator $\Phi$ in property (2), and the operator $\Phi$ in Proposition 2.2.18.) This criterion allows us to verify that a concretely defined “guess” is indeed the initial algebra, and is not too difficult to verify in practice. Some examples are given in [FM91]; in principle, this method allows us to check that the free $\Delta$-algebra $T_e X$ is as expected, consisting of syntactic effect-trees $\delta((x_i)_{i \in I})$ with the ordering described in the previous section: one tree is above another if and only if it is obtained by replacing any $x_i$ with $x'_i$ such that $x_i \sqsubseteq x'_i$, or occurrences of $\perp$ with new sub-trees.

We now consider how to prove that $T_e$ is a $\text{Cpo}_{\perp!}$-monad, in order to guarantee its strictness, and hence left-strictness of composition in $\text{Kl}(T_e)$.

**Remark 3.3.10.** To do this, one may be tempted to express $T_e$ as a monad $U^\Delta F^\Delta$ arising from an explicit adjunction, shown below and following Section 2.1.3, and show that the left-adjoint $F^\Delta$ of $U^\Delta$ is a $\text{Cpo}_{\perp!}$-functor (as $U^\Delta$ trivially is), by exploiting a connection between ordinary and enriched adjoints.

$$
\begin{array}{ccc}
\text{Cpo}_{\perp!} & \xrightarrow{F^\Delta} & \text{Alg}(\Delta) \\
\perp & \downarrow & \downarrow U^\Delta \\
\end{array}
$$

However, as far as we know, the only relevant result in this direction is in [Kel05] (pp. 24): A $\mathcal{V}$-functor $U$ between two $\mathcal{V}$-categories has a $\mathcal{V}$-enriched left adjoint whenever it has an ordinary left-adjoint, if the functor $V = \mathcal{V}_0(I, -) : \mathcal{V} \to \text{Set}$ is conservative (i.e. isomorphism-reflecting: whenever $Vg$ is an isomorphism, so is $g$). Unfortunately, for $\mathcal{V} = \text{Cpo}_{\perp!}$, the functor $V$ is not conservative. Consider the following cpos $A, B$, and function $f$ between them; it is not an...
isomorphism in $\mathsf{Cpo}_{\bot}$, but its image $Vf$ is an isomorphism of sets.

$$
\begin{array}{ccc}
A : & a & b \\
& \downarrow & \\
\bot & & \\
\end{array}
\begin{array}{ccc}
B : & d & c \\
& \downarrow & \\
\bot & & \\
\end{array}
\begin{array}{ccc}
f : a \mapsto c, & b \mapsto d, & \bot \mapsto \bot \\
\end{array}
$$

Instead, we adopt a more elementary approach, exploiting the fact that for an arrow $f : X \to Y$, the map $Tef : T_eX \to T_eY$ is the unique arrow $!$ making the following diagram commute (Section 2.1.2), where we write $\psi_X$ for the $\Delta$-algebra structure of $T_eX$, and similarly for $Y$ (we use the same notation $\psi_X$ for the $\Sigma$-algebra structure of the syntax terms $TX$, and let the context determine which is meant).

$$
\begin{array}{ccc}
\Delta T_eX & \xrightarrow{\Delta!} & \Delta T_eY \\
\downarrow \psi_X & & \downarrow \psi_Y \\
T_eX & \xrightarrow{!} & T_eY \\
\eta_X & & \eta_Y \\
X & \xrightarrow{f} & Y \\
\end{array}
$$

To prove that $T_e$ is $\mathsf{Cpo}_{\bot}$-enriched, we note that the functors $\oplus, \times, \otimes$ are $\mathsf{Cpo}_{\bot}$-enriched, and hence so are the syntax functors $\Delta$ built from composing these functors. This allows us to exploit the following result:

**Lemma 3.3.11.** In the category $\mathsf{Cpo}_{\bot}$, if an endofunctor $\Delta$ is locally continuous and strict (i.e. $\mathsf{Cpo}_{\bot}$-enriched), then so is the free $\Delta$-algebra functor $T_e$.

**Proof.** Strictness means that $T_e \bot X, Y = \bot T_e X, T_e Y$; to prove this, we show that $\bot T_e X, T_e Y = !$ makes the above diagram commute, where we take $f = \bot X, Y$. It is broken down as follows, where the top equality is by strictness of $\Delta$. Each triangle commutes as composition in $\mathsf{Cpo}_{\bot}$ is strict, and the paths contain bottom arrows $\bot_{P,Q}$.

$$
\begin{array}{ccc}
\Delta T_eX & \xrightarrow{\Delta \bot T_e X, T_e Y} & \Delta T_eY \\
\downarrow \psi_X & & \downarrow \psi_Y \\
T_eX & \xrightarrow{\bot T_e X, T_e Y} & T_eY \\
\eta_X & & \eta_Y \\
X & \xrightarrow{\bot X, Y} & Y \\
\end{array}
$$

To prove local continuity, we need to show that for an $\omega$-chain of arrows $(f_n : X \to Y)_{n<\omega}$, we have $T_e \sqcup_{n<\omega} f_n = \sqcup_{n<\omega} (T_e f_n)$. Again, we show that $! = \sqcup_{n<\omega} (T_e f_n)$ makes the same diagram
commute, where the equality is by local continuity of $\Delta$.

$$\Delta T_e X \xrightarrow{\Delta(\sqcup_{n<\omega} T_e f_n)} \Delta T_e Y$$

Now the top half commutes by the following reasoning. The first and last steps are by continuity of composition in $\mathsf{Cpo}_{\bot!}$ in both arguments; the second is by naturality of $\psi : \Delta T_e \Rightarrow T_e$. The bottom half commutes by a similar argument, using naturality of $\eta : \text{Id} \Rightarrow T_e$.

$$\psi_Y \circ \sqcup_{n<\omega}(\Delta T_e f_n) = \sqcup_{n<\omega}(\psi_Y \circ \Delta T_e f_n) = \sqcup_{n<\omega}(T_e f_n \circ \psi_X) = (\sqcup_{n<\omega} T_e f_n) \circ \psi_X$$

By contrast, the free program-syntax functor $T$ will generally not be $\mathsf{Cpo}_{\bot!}$-enriched, as it will not be strict. To build syntax terms of a closed language, one must have some constants; and this means that $\Sigma$ will not be strict, so the previous lemma will not apply (on replacing $\Delta$ with $\Sigma$ and $T_e$ with $T$). As an example, taking $\Sigma X = 1_\bot \oplus X$, one has the $\Sigma$-algebra $A = (1_\bot, \gamma = [\text{id}, \text{id}] : 1_\bot \oplus 1_\bot \rightarrow 1_\bot)$. It is easy to check that the bottom arrow $\bot$ is not a $\Sigma$-algebra morphism from $A$ to itself: the following diagram does not commute, as the left-hand occurrence of $1_\bot$ is sent to different places by both paths.

$$\begin{array}{ccc}
1_\bot + 1_\bot & \xrightarrow{1_\bot \oplus 1_\bot} & 1_\bot + 1_\bot \\
\downarrow{[\text{id}, \text{id}]} & \times & \downarrow{[\text{id}, \text{id}]} \\
1_\bot & \xrightarrow{\bot} & 1_\bot
\end{array}$$

**Final Kleisli-Coalgebra Semantics, Concretely**

This allows us to conclude that the Kleisli category $\mathsf{Kl}(T_e)$ for syntactic effects has all the order structure required to guarantee existence of a final Kleisli coalgebra. In the previous section, we showed that it inherits $\mathsf{Cpo}_{\bot!}$-enrichedness from the underlying category $\mathsf{Cpo}_{\bot!}$; we similarly defined a behaviour functor $B X = V + X$ in terms of a categorical coproduct, which we instantiated in $\mathsf{Cpo}_{\bot!}$ by $B X = \mathsf{Vals}_\bot \oplus X$. The functor is locally continuous and has a lifting $\overline{B}$ to any Kleisli category, given by the distributive law $\lambda_X$ of Example 2.2.10. As the effect-syntax functor $\Delta$ is built from smash products and coproducts, it is a $\mathsf{Cpo}_{\bot!}$-functor, and hence so is $T_e$ by Lemma 3.3.11, ensuring that Kleisli-composition is left-strict.
Corollary 3.3.3 now holds, so that the final $\overline{B}$-coalgebra is given by the initial $B$-algebra $\overline{D}$, which exists by algebraic $\omega$-compactness. In the setting of $\text{Cpo}_{\perp\bot}$, it may be shown concretely, following the method of Remark 3.3.8, to be the $N$-fold coproduct $N \cdot \text{Vals}_\bot$ of the flat cppo $\text{Vals}_\bot$. This is another flat cppo, which consists of pairs $(n, v)$ of a natural number $n$ and a value $v$ in (one of the sort-components of) $\text{Vals}$, which we underline.

Thus, for every syntactic $\text{ets}$ – a $T_eB$-coalgebra $(X, \gamma)$, or equivalently a $\overline{B}$-coalgebra $(X, \gamma)$ (omitting the decoration ‘on $\gamma$’) – there is a canonical $B$-coalgebra morphism $\overline{\beta}_\gamma : X \to N \cdot \text{Vals}_\bot$, of underlying type $X \to T_e(N \cdot \text{Vals}_\bot)$. We anticipated that the map $\overline{\beta}_\gamma$ should assign to each coalgebra-state $x$ the syntactic effect-tree $\delta((b_i)_{i \in I})$ corresponding to its execution, with leaves $b_i$ (in $N \cdot \text{Vals}_\bot$) given either by $\perp$ (for divergent computation branches), or otherwise a pair $(n, v)$ of the steps-to-termination $n$ and the terminal value $v$ produced by that branch. We illustrate how this may be checked concretely, as Proposition 2.2.18 characterises the final coalgebra morphism $\overline{\beta}_\gamma : X \to \overline{D}$ as a join of approximants $\overline{\beta}^{(n)}$, which are straightforward to calculate. (As shown, we omit the subscript $\gamma'$ on approximants $\overline{\beta}^{(n)}$ for readability.)

**Example 3.3.12.** We assume one has a syntactic $\text{ets}$ $(T0, \text{om})$, where $\text{om}$ is of type $T0 \to T_e(\text{Vals}_\bot \oplus T0)$, corresponding to an operational model for While programs. (We will show how to do this in the following section.) On this basis, we illustrate how the fixpoint construction of Proposition 2.2.18 assigns effect-trees in the semantic domain $T_e(N \cdot \text{Vals}_\bot)$ to While programs.

Taking $BX = \text{Vals}_\bot \oplus X$, the initial $B$-algebra $\overline{D}$ has carrier $N \cdot \text{Vals}_\bot$ as described above, and its algebra-structure $\alpha : B\overline{D} \to \overline{D}$ is defined by $\alpha(\text{inl}(v)) = (1, v)$ and $\alpha(\text{inr}(n, v)) = (n + 1, v)$ (and $\alpha(\perp) = \perp$).

We illustrate the action of the approximants $\overline{\beta}^{(n)}$ on the programs $x = n$ for any natural $n$, and the program $x = 5; x = 8$. (We abbreviate the series of maps $(B\overline{\beta}^{(0)})^{\dagger} = \mu^{T_e} \circ T_e \lambda \circ T_e B\overline{\beta}^{(0)}$, which have no effect on the terminated value $\text{inl}(\ast)$.)

\[
\overline{\beta}^{(1)} : \quad x = n \xrightarrow{\text{om}} \text{wr}_{x,n}(\text{inl}(\ast)) \xrightarrow{(B\overline{\beta}^{(0)})^{\dagger}} \text{wr}_{x,n}(\text{inl}(\ast)) \xrightarrow{T_e \alpha} \text{wr}_{x,n}(1, \ast)
\]

This yields the denotation of the assignment $x = n$: an effect-tree with a single leaf, describing termination after 1 step with void return-value $\ast$. Note that the join $\overline{\beta}$ of $\overline{\beta}^{(1)}$ with higher approximants $\overline{\beta}^{(n)}$ – whose definition shows they form a $\omega$-chain – must also assign the same denotation to $x = n$, as there is no element above it in the semantic domain $T_e\overline{D}$. We may now calculate the denotation of the program $x = 5; x = 8$. (Below, the multiplication $\mu^{T_e}$ ignores the formal distinction between the two layers of effect-syntax, $\text{wr}_{x,5}$ and $\text{wr}_{x,8}$.)
\[
\overline{\beta}^{(2)}: (x = 5; x = 8) \xrightarrow{\mathrm{om}} \text{wr}_{x,5}(\mathrm{inr}(x = 8)) \\
\xrightarrow{T_eB\overline{\beta}^{(1)}} \text{wr}_{x,5}(\mathrm{inr}(\text{wr}_{x,8}(1, *)) ) \\
\xrightarrow{T_e\lambda} \text{wr}_{x,5}(\text{wr}_{x,8}(\mathrm{inr}(1, *))) \\
\xrightarrow{\mu_T} \text{wr}_{x,5}(\text{wr}_{x,8}(\text{wr}_{x,8}(\mathrm{inr}(1, *)) ) ) \\
\xrightarrow{T_e\alpha} \text{wr}_{x,5}(\text{wr}_{x,8}(2, *)) 
\]

In this way, the final coalgebra morphism \( \overline{\beta}_{\mathrm{om}} \) assigns to each program the overall effect-tree observed during its execution, and the leaves of the effect-tree describe its execution traces: the number of steps to termination and the return value; or \( \bot \) for divergent branches.

### 3.3.4 From Effectless to Effectful Operational Specifications

Having defined a semantic domain \( T_eD \) for effectful programs, we now consider the problem of specifying an operational model \( \langle T0, \text{om} : T0 \to T_eBT0 \rangle \) in the form of a syntactic \( eTS \), i.e. a \( T_eB \)-coalgebra. As before, we aim to achieve this by structural recursion (Proposition 2.1.26), given an abstract operational semantics. In the context of \( T_eB \)-coalgebras, this is a natural transformation \( \epsilon_X : \Sigma(X \times T_eBX) \to T_eBTX \).

The problem is that such specifications are unconventional and difficult to define directly, as we saw this in the context of While programs in Section 3.1.2. By contrast, it is relatively straightforward to specify the effectless fragments of language, such as \( S\text{While} \), as shown in Example 3.1.7. Thus, it is natural to ask if effectless specifications may be extended to incorporate effects. In this section, we show how a suitable restriction on effectless specifications – given by natural transformations \( \rho_X : \Sigma(X \times BX) \to BTX \) – allows us to extend them to specifications for an effectful language.

One must begin by deciding how to extend the effect-free fragment of a language syntactically. The simplest strategy is to introduce the effect-syntax into the language directly; this means extending the syntax signature \( \text{Sig} \) with the (single-sorted) effect signature \( \text{Eff} \), at every sort. We introduce such an extension of \( S\text{While} \): as shown below, we underline effect-syntax \( e \) in programs, as opposed to semantic effects \( e \) observed during execution. (This is the opposite of our convention for syntactic values \( v \) and semantic return values \( v \).)

**Definition 3.3.13.** Given a set of variable locations \( L \), the language \( s\text{EWhile} \), or ‘Syntactic Effectful While’, has all the syntax constructors of \( S\text{While} \), and in addition, for each sort \( s \) and all \( x \in L, n \in \mathbb{N} \), it has a unary operator \( \text{wr}_{x,n} : s \to s \) and an \( \mathbb{N} \)-ary operator \( \text{rd}_x : s^\mathbb{N} \to s \).

An example of an \( s\text{EWhile} \) program is \( \text{wr}_{x,5}(\text{skip}); \text{wr}_{x,8}(\text{skip}) \), which we expect to behave as follows:
Functorially, this syntactic extension amounts to replacing the program-syntax functor $\Sigma$ with $\Sigma' = \Sigma + \Delta$. We correspondingly write $T'X$ for the free $\Sigma'$-algebra monad.

An alternative strategy, rather than introducing effect-syntax directly into the language, is to introduce ‘custom’ commands $\Sigma_2$ for specific effect-trees; we comment on this strategy at the end of this section.

The next step is to extend an abstract OS for the effect-free fragment of the language, given by a natural transformation $\rho_X : \Sigma(X \times BX) \rightarrow BTX$, into one for the full language, of type $\epsilon_X : \Sigma'(X \times T_{e}BX) \rightarrow T_{e}BT'X$. As before, this is a specification in the following sense: given a syntax term $\Sigma'$ with arguments $X$ and their effectful behaviours $T_{e}BX$, the map tells us the effectful behaviour of that term, having effectful transitions $T_{e}B$ to arbitrary terms $T'X$ (allowed to contain both program and effect syntax, although we never introduce the latter).

The domain of $\epsilon_X$ is a coproduct, as $\Sigma' = \Sigma + \Delta$; thus it may be defined by two separate specifications

$$\epsilon_X^{(1)} : \Delta(X \times T_{e}BX) \rightarrow T_{e}BT'X \quad \epsilon_X^{(2)} : \Sigma(X \times T_{e}BX) \rightarrow T_{e}BT'X$$

where the first handles effect-syntax in programs, and the second specifies how syntax operators in $\Sigma$ interact with effectful behaviour. The first is straightforward, essentially represented by trivial rules such as:

$$\text{wr}_{x,5}(p) \rightarrow \text{wr}_{x,5}(p)$$

This means that when a syntactic effect $e$ is encountered in a program $p = e((p_i)_{i \in I})$, that program exhibits the effect $e$ in its behaviour; the computation-branches undergo transitions to $p_i$. The corresponding specification $\epsilon_X^{(1)}$ is given by the following composition (where $\psi^\Delta_Y$ is the $\Delta$-algebra structure $\Delta T_{e}Y \rightarrow T_{e}X$ of the free $\Delta$-algebra $T_{e}Y$; and $\eta^{T''}$ is the unit of the free $\Sigma'$-algebra monad $T'$.)

$$\epsilon_X^{(1)} : \Delta(X \times T_{e}BX) \xrightarrow{\Delta \pi_1} \Delta X \xrightarrow{\Delta \eta^\Delta_X} \Delta T_{e}X \xrightarrow{\psi^\Delta_X} T_{e}X \xrightarrow{T_{e}B \eta^{T''}} T_{e}BT'X.$$  

The more substantial difficulty is in the second specification $\epsilon_X^{(2)}$. To illustrate, we consider how the program $\text{if} \ (e) \ \text{then} \ \{p\} \ \text{else} \ \{q\}$ should behave when $e, p, q$ exhibit effects. For instance, we might have

$$e \rightarrow \text{rd}_x(\text{true}, e', \text{false}, \ldots)$$

with the intended meaning that $e$ will perform a lookup on variable $x$, and behave according
to its value: returning value \texttt{true} if \( x \) is 0, transitioning \( e \to e' \) if \( x \) is 1, returning \texttt{false} if \( x \) is 2, and so on. We would expect the behaviour of \( \text{if}(e)\text{then}\{p\}\text{else}\{q\} \) to depend similarly on the value of \( x \) – the next transition would be to the state \( p \) if \( x = 0 \); it would be \( \text{if}(e')\text{then}\{p\}\text{else}\{q\} \) if \( x = 1 \); and it would be \( q \) if \( x = 2 \) (and so on). Thus it should have the transition

\[
\text{if}(e)\text{then}\{p\}\text{else}\{q\} \to \text{rd}_x(p, \text{if}(e')\text{then}\{p\}\text{else}\{q\}), q, \ldots).
\]

This means \textit{applying the operational rule for \texttt{if} at each branch of the behaviour of \( e \).}

Now we consider how this information appears in categorical terms. As described in Section 3.3.3, we assume each argument \( x_i \) to a syntax-constructor \( \sigma(x_1, \ldots) \) is represented by a suitable product \( X \times (-) \) or \( X \otimes (-) \) within a larger product (e.g. \( X \times (X \otimes (X \times X)) \)). The behaviour of this argument is given by an element of \( T_e BX \), consisting of an effect-syntax tree with leaves in \( BX \); to apply operational rules at each leaf of the tree, we need to attach information to each branch. For instance, if \( e \to \text{rd}_x(\text{true}, e', \text{false}, \ldots) \) as above, and we wish to determine the behaviour of the statement \( \text{if}(e)\text{then}\{p\}\text{else}\{q\} \), then we would have to attach the arguments \( p, q \) to each branch of this effect-tree, informally giving

\[
e \to \text{rd}_x(\text{true}, p, q), (e', p, q), (\text{false}, p, q), \ldots)
\]

and this gives us the information we need at each leaf for applying the effect-free operational rules for \texttt{if} statements.

Representing this added information by an object \( Y \), the above operation would amount to a map \( T_e BX \times Y \to T_e(BX \times Y) \) if we represented the argument by a Cartesian product \( \times \), otherwise a map \( T_e BX \otimes Y \to T_e(BX \otimes Y) \) if we used the smash product; and these maps resemble \textit{monadic (co)strength}. Recall that the single-sorted monad \( T_e 0 \), and hence the multi-sorted version \( T_e \), are \( \text{Cpo}_4 \)-enriched by our assumptions on \( \Delta_0 \); and this means it has a costrength \( \text{cost}_{P,Q} : T_e P \otimes Q \to T_e(P \otimes Q) \) (and a strength!), as described in Section 2.1.4. (We later make use of the costrength of \( T_e 0 \), which we call \( \text{cost}^{(0)} \).)

We will use the costrength to propagate information \( Q \) to branches \( P \) of an effect-tree \( T_e P \). By contrast, in \( \text{Cpo}_4 \), one finds there is no suitable candidate for a ‘costrength with respect to Cartesian products’, i.e. a natural transformation \( T_e P \times Q \to T_e(P \times Q) \).

This leads to a restriction on the program-syntax functor \( \Sigma \), and a restriction on the effect-free specifications which can be extended with effects: firstly, \textit{if the behaviour of an argument \( x_i \) plays a role in the behaviour of the term \( \sigma(x_1, \ldots) \), then that argument should be represented functorially by the monoidal (smash) product \( X \otimes (-) \)}. We refer to such arguments as ‘active’ arguments. For instance, given a collection \( (X_N, X_E, X_P) \) of \( N, E, P \)-sorted syntax variables \( X \) in \( \text{Cpo}_4^3 \), we should represent the boolean-sort argument \( X_E \) of an \texttt{if} statement by a smash
product $X_E \otimes (-)$. (The other arguments are considered shortly.) Without loss of generality, we may suppose that each syntax constructor $\sigma$ depends on the behaviour of the first $m$ arguments, for some $0 \leq m < \text{ar}(\sigma)$. We formalise this by a dependency function:

**Definition 3.3.14.** For a signature $\text{Sig}$ of syntax symbols $f$, a dependency function is a function $\text{dep} : \text{Sig} \rightarrow \mathbb{N}$ satisfying $0 \leq \text{dep}(f) \leq \text{ar}(f)$ for every symbol $f \in \text{Sig}$.

When defining an effectful extension of an effect-free specification, the correct choice of dependency function will ensure that we do not introduce effects which should not (yet) be observed. For instance, if we asserted that $\text{dep}(\text{if}) = 3$ for $\text{if}$ statements, it would mean that the transition behaviour of $\text{if}(e) \text{then} \{p\} \text{else} \{q\}$ would introduce whatever effects are produced by $p$ and $q$, which is wrong: it would amount to incorrect derivations such as

$$
eq \rightarrow \text{rd}_{x}(\text{true}, e', \ldots) \quad p \rightarrow \text{wr}_{x,5}(p') \quad q \rightarrow \text{wr}_{y,6}(\ast) \quad \text{if}(e) \text{then} \{p\} \text{else} \{q\} \rightarrow \text{rd}_{x}(\text{wr}_{x,5}(\text{wr}_{y,6}(p)), \text{wr}_{x,5}(\text{wr}_{y,6}(\text{if}(e') \text{then} \{p\} \text{else} \{q\})), \ldots)
$$

The correct value would be $\text{dep}(\text{if}) = 1$. By contrast, when executing a $\text{while}$ loop, its transition behaviour (to a suitable $\text{if}$ statement, as illustrated in Section 3.1.2) happens independently of the behaviour of any arguments; and hence we would take $\text{dep}(\text{while}) = 0$.

Another restriction comes from the fact that effectful abstract OS specifications $\epsilon_X$ are assumed to be given in terms of categorical (i.e. in $\text{Cpo}_{\perp!}$, Cartesian) products, with codomain $\Sigma(X \times T_e B X)$. This implies that the derived behaviour of a term $\sigma(e, \ldots)$ cannot refer to the ‘current’ value $e$ of any active arguments, but only on their successors $e'$ (if $e \rightarrow e'$) or terminal values $v$ (if $e \rightarrow v$). For instance, the operational rules for $\text{if}(e) \text{then} \{p\} \text{else} \{q\}$ involve transitions either to $\text{if}(e') \text{then} \{p\} \text{else} \{q\}$, $p$, or $q$, and there is no reference to the ‘current’ value $e$ of the active argument, so they satisfy this restriction.

The reason this restriction is needed is that if the effect-free operational rules of a term depended on the current value, then to apply those rules at each leaf of an effect-tree, we would again need a strength-like natural transformation $X \times T_e Y \rightarrow T_e (X \times Y)$ for categorical products, which need not exist in general. We will shortly formalise this restriction on the effect-free specifications.

There is a need for a further restriction on the syntax functor $\Sigma$, involving categorical products $X \times Y$. These have canonical projections $\pi_1 : (X \times Y) \rightarrow X$ and similarly $\pi_2$ for $Y$, which are natural in both arguments; this is necessary when defining abstract OS specifications to recombine the arguments of syntax terms $\sigma(x_1, \ldots)$ into new terms. For instance, in the context of $\text{Set}$ we described an $\text{if}$ statement as a tuple in the categorical product $X_E \times X_P \times X_P$ in Example 3.1.7. When considering its behaviour, the specification of a transition like $\text{if}(e) \text{then} \{p\} \text{else} \{q\} \rightarrow p$ (when $e \rightarrow \text{true}$) referred to projections $\pi_1 : (X_P \times X_P) \rightarrow X_P$; and similarly $\pi_2$ for specifying $\text{if}(e) \text{then} \{p\} \text{else} \{q\} \rightarrow q$. However, in $\text{Cpo}_{\perp!}$, if we represent
the two branches of the if statement using a smash product $X_E \otimes (X_P \otimes X_P)$, rather than the categorical Cartesian product $X_E \otimes (X_P \times X_P)$, we will find that there is no categorically natural way to ‘project out’ the components of a smash product:

**Lemma 3.3.15.** In $\mathsf{Cpo}_\bot !$, the only natural transformation $\alpha_{X,Y} : (X \otimes Y) \to X$ natural in $Y$, is given uniformly by $\bot$-valued maps $\bot_{(X \otimes Y),X}$.

**Proof.** Naturality in $Y$ requires that for all arrows $f : Y \to Y'$, the following diagram commutes.

\[
\begin{array}{ccc}
(X \otimes Y) & \xrightarrow{\alpha_{X,Y}} & X \\
(id \otimes f) \downarrow & & \downarrow id \\
(X \otimes Y') & \xrightarrow{\alpha_{X,Y'}} & X
\end{array}
\]

If we take $Y = Y'$ and $f = \bot_{Y,Y}$, the diagram implies the following, where the second step uses the fact that the bifunctor $\otimes$ is strict in both arguments (or, by its concrete definition in terms of pairs, because $(x, \bot) = \bot$), and the third is by strictness of composition in $\mathsf{Cpo}_\bot !$.

\[
\alpha_{X,Y} = \alpha_{X,Y} \circ (id \otimes \bot_{Y,Y}) = \alpha_{X,Y} \circ \bot_{X \otimes Y, X \otimes Y} = \bot_{(X \otimes Y), X}
\]

Thus, if we wish to construct a new syntax term $t_2((y_j)_{j \in J})$ out of an old term $t_1((x_i)_{i \in I})$ where $\{y_j : J \in J\} \subseteq \{x_i : i \in I\}$, we must make sure the relevant arguments of $t_1$ are given by categorical products, in order to recombine them with projections and inclusions. In this recombination process, when required one may convert a categorical product $X \times Y$ into a smash product $X \otimes Y$ as follows, using the following natural transformation. (Here, given a cppo $X$, in $X_\bot$ we write $\bot$ for the original bottom element of $X$, and $\bot'$ for the extra bottom element attached by the functor $(-)_\bot$.)

\[
p_X : X_\bot \to X \quad x \mapsto x, \ \bot \mapsto \bot, \ \bot' \mapsto \bot
\]

We may post-compose the natural isomorphism $(X \times Y) \cong (X_\bot \otimes Y_\bot)$ with $(p_X \otimes p_Y)$ to obtain a natural transformation $(X \times Y) \to (X \otimes Y)$. By contrast, in the other direction one finds the obvious candidate for a natural transformation $m_X : X \to X_\bot$ is not in fact natural.

These restrictions lead us to the following formal assumption on syntax functors $\Sigma$ and abstract OS specifications $\epsilon$:

**Definition 3.3.16.** For a signature $\mathsf{Sig}$ with dependency function $\mathsf{dep}$, the syntax functor $\Sigma$
on \( \text{Cpo}_{\perp\perp} \) defined by \( \text{Sig} \) and \( \text{dep} \) is as follows:

\[
(\Sigma X)_s = \prod_{\sigma(s_i) : 0 \leq i < \text{car}(\sigma) \rightarrow s} \left( \bigotimes_{0 \leq i < \text{dep}(\sigma)} X_{s_i} \right) \otimes \left( \bigotimes_{\text{dep}(\sigma) \leq i < \text{car}(\sigma)} X_{s_i} \right)
\]

An effect-free abstract OS specification with respect to \( \text{Sig} \) and \( \text{dep} \), viz. \( \epsilon_X : \Sigma(X \times BX) \rightarrow BTX \), is one expressible as follows, for some natural transformation \( \rho \). (To save space, we shorten the subscript of each coproduct from \( \sigma : (s_i)_{0 \leq i < \text{car}(\sigma)} \rightarrow s \) to just \( \sigma \).)

\[
(\Sigma(X \times BX))_s = \prod_{\sigma} ((\bigotimes_{0 \leq i < \text{dep}(\sigma)} (X \times BX)_{s_i}) \otimes (\bigotimes_{\text{dep}(\sigma) \leq i < \text{car}(\sigma)} ((X \times BX)_{s_i})))
\]

\[
= \prod_{\sigma} ((\bigotimes_{0 \leq i < \text{dep}(\sigma)} (X_{s_i} \times (BX)_{s_i}) \otimes (\bigotimes_{\text{dep}(\sigma) \leq i < \text{car}(\sigma)} (X_{s_i} \times (BX)_{s_i})))
\]

\[
\xrightarrow{\text{LL}_i \otimes (\pi_2 \otimes \prod_{\pi_1})} \prod_{\sigma} ((\bigotimes_{0 \leq i < \text{dep}(\sigma)} ((BX)_{s_i}) \otimes (\bigotimes_{\text{dep}(\sigma) \leq i < \text{car}(\sigma)} X_{s_i})))
\]

Example 3.3.17. With \( \Sigma \) in this restricted form, we illustrate part of the specification of an \( \text{if } (e) \text{ then } \{ p \} \text{ else } \{ q \} \) statement, by taking \( \text{Sig} \) to be \( \{ \text{if } : E, P, P \rightarrow P \} \) and \( \text{dep}(\text{if}) = 1 \), so that \( (\Sigma X)_P = X_E \otimes X^2_P \). We also take \( (BX)_s = V_s + X_s \), where the cppo of boolean return values \( V_E \) is given by \( (\mathbb{B})_{\perp} \cong 1_{\perp} + 1_{\perp} \). This specification is closely analogous to Example 3.1.7, but adapted to the setting of \( \text{Cpo}_{\perp\perp} \); we give more details below.

\[
(\Sigma(X \times BX))_P = (X_E \times (BX)_E) \otimes (X_P \times (BX)_P)^2
\]

\[= (X_E \times (V_E + X_E)) \otimes (X_P \times (V_P + X_P))^2\]

\[\xrightarrow{\pi_2 \otimes (\pi_1)^2} (V_E + X_E) \otimes X^2_P\]

\[= ((1_{\perp} + 1_{\perp}) + X_E) \otimes X^2_P\]

\[\xrightarrow{\text{dist} (1_{\perp} + 1_{\perp}) \cdot X_E \cdot X^2_P} (1_{\perp} + 1_{\perp}) \otimes X^2_P + X_E \otimes X^2_P\]

\[\xrightarrow{\text{dist} 1_{\perp} \cdot X^2_P + \text{id}} (1_{\perp} \otimes X^2_P) + (1_{\perp} \otimes X^2_P) \otimes X_E \otimes X^2_P\]

\[\xrightarrow{\text{id}} X^2_P + X^2_P + X_E \otimes X^2_P\]

\[\xrightarrow{\pi_1 + \pi_2 + \text{id}} X_P + X_P + X_E \otimes X^2_P\]

Above, the third line isolates (via \( \pi_2 \)) the behaviour \( V_E + X_E \) of the active argument \( e \), and discards the irrelevant behaviours (via \( \pi_1 \)) of the other arguments \( p \) and \( q \). The next two lines use (silently reversed) distributivity isomorphisms to distinguish the cases where \( e \rightarrow \text{true}, \) \( e \rightarrow \text{false} \), and \( e \rightarrow e' \) – which correspond to the three coproduct components on the seventh line, after cleaning up using monoidal isomorphisms \( 1_{\perp} \otimes Y \cong Y \). The last line uses categorical projections \( \pi_1, \pi_2 \) to return \( p \) if \( e \rightarrow \text{true}, \) \( q \) if \( e \rightarrow \text{false}, \) and a new \( \text{if} \) statement if \( e \rightarrow e' \). (We omit the rest of the the specification, which is mostly bureaucratic detail.)

This approach gives a canonical means of defining the effectful specification \( \epsilon^{(2)} \) of effect-free syntax constructors if the behaviours of every program term depends on that of at most one
Chapter 3. MOS in a Mixed Kleisli Category for Syntactic Effects

subterm – i.e. \( \text{dep}(\sigma) \leq 1 \) for all \( \sigma \). However, two issues arise if \( \text{dep}(\sigma) > 1 \). Consider a ‘synchronous execution’ operator for command-type expressions:

\[
\begin{align*}
  p \rightarrow p', q \rightarrow q' & \quad p \rightarrow p', q \rightarrow * & \quad p \rightarrow *, q \rightarrow q' & \quad p \rightarrow *, q \rightarrow *
  \\
  p \times q \rightarrow p' \times q' & \quad p \times q \rightarrow p' & \quad p \times q \rightarrow q' & \quad p \times q \rightarrow *
\end{align*}
\]

Although the operational semantics of \( p \times q \) is trivial, its effectful extension is less so: If \( p \) and \( q \) both introduce effects, such as writing to the same variable, obviously there can be no canonical choice for the effectful behaviour of \( p \times q \); one must make a choice whether to apply the variable update of \( p \) first, or of \( q \). Hence, we are forced to put an ordering on the sub-terms specifying the order of propagation of effects. Without loss of generality, we will suppose this ordering is simply left-to-right, and define a natural transformation \( \text{comb} \) to propagate effects in this way using monadic strength.

The second issue is that a term may not always depend on the same number of sub-terms. In \textbf{While}, \( + \) and \( * \) are examples; without auxiliary operators \( +_n \) and \( *_n \), a standard operational semantics might contain the rules

\[
\begin{align*}
  u \rightarrow u' & \quad u \rightarrow 0, v \rightarrow v' & \quad u \rightarrow 0, v \rightarrow n \rightarrow m & \quad u \rightarrow n, v \rightarrow v \\
  u + v \rightarrow u' + v & \quad u + v \rightarrow n + v' & \quad u + v \rightarrow n + m
\end{align*}
\]

In applying the first rule, we must not propagate any effects of \( v \), unlike the other cases. This would require a more fine-grained approach than the one we have considered; however, introducing auxiliaries can ensure the behaviour of every syntax constructor once again always depends on the same sub-terms, as indicated by dependency functions \( \text{dep} \). This is why we introduced the operators \( +_n \) and \( *_n \) in our syntax for \textbf{While}.

Under the assumption that the behaviour of each syntax constructor \( \sigma \) always depends on the same number of arguments \( \text{dep}(\sigma) \), and that their effects are to be propagated from left to right, we now define functions \( \text{comb}_{Y_0, \ldots, Y_{n-1}} \) which propagate effects from \( Y_0, \ldots, Y_{n-1} \) by applying monadic strength repeatedly. We will need to do this at each sort, and so the definition is essentially single-sorted. (We include a definition \( \text{comb} \) for the case where there are no arguments, given by an empty smash product which we consider to be \( 1_\bot \); both this, and the indexing of \( Y_0, \ldots, Y_{n-1} \) from 0, are for convenience in the following definition of effectful extensions.)

\textbf{Definition 3.3.18.} Let \( M \) be a \( \otimes \)-strong monad on \( \text{Cpo}_\bot \). Taking the empty smash product to be \( 1_\bot \), we define \( \text{comb} = \eta_{1_\bot} : 1_\bot \rightarrow M1_\bot \). Furthermore, for all \( 1 \leq n < \omega \), given objects \( Y_0, \ldots, Y_{n-1} \), we inductively define arrows \( \text{comb}_{Y_0, \ldots, Y_{n-1}} : \bigotimes_{0 \leq i < n} (M_\bot i) \rightarrow M \left( \bigotimes_{0 \leq i < n} Y_i \right) \) by \( \text{comb}_{Y_0} = \text{id} \), and then \( \text{comb}_{Y_0, \ldots, Y_n} \) in terms of \( \text{comb}_{Y_0, \ldots, Y_{n-1}} \) as follows. (We omit some trivial

\footnote{If one were to impose equations on the effects in \( T_e \), giving rise to a \textit{commutative} monad – such as the equations for non-determinism, giving the finite powerset monad \( P_\bot \) – then this choice would make no difference and the effectful extension would be canonical; but otherwise, one would have to impose such an ordering.}
isomorphisms, and abbreviate the costrength \( \text{cost(x)}_{\otimes Y_i, MY_n} \) and then the strength \( \text{st(x)}_{\otimes Y_i, Y_n} \).

\[
\begin{align*}
\text{cost} & \rightarrow M(\bigotimes_{0 \leq i < n} Y_i) \otimes MY_n \\
\mu \otimes Y_i & \rightarrow M(\bigotimes_{0 \leq i < n} Y_i) \otimes MY_n
\end{align*}
\]

Finally, we suppose we are given a signature \( \text{Sig} \) and dependency functor \( \text{dep} \), with corresponding syntax functor \( \Sigma \) and an effect-free abstract OS specification in terms of a natural transformation \( \rho \) as in Definition 3.3.16. We then define the \( s \)-component of the effectful specification \( \epsilon_X^{(2)} \) by the following composition, explained in more detail below.

\[
\begin{align*}
\epsilon_X^{(2)} : \Pi_{s_{\sigma}} &\rightarrow \Pi_{\sigma : (s_i) \rightarrow s} \left( \bigotimes_{0 \leq i < \text{dep}(\sigma)} (X \times T_e BX)_{s_i} \otimes \prod_{\text{dep}(\sigma) \leq i < \text{ar}(\sigma)} (X \times T_e BX)_{s_i} \right) \\
\eta_{\pi_2 \Pi(\pi_1)} &\rightarrow \Pi_{\sigma : (s_i) \rightarrow s} \left( \bigotimes_{0 \leq i < \text{dep}(\sigma)} (T_e BX)_{s_i} \otimes \prod_{\text{dep}(\sigma) \leq i < \text{ar}(\sigma)} X_{s_i} \right) \\
\Pi_{\text{cost}(0)} &\rightarrow \Pi_{\sigma : (s_i) \rightarrow s} \left( T_{e \sigma} \left( \bigotimes_{0 \leq i < \text{dep}(\sigma)} (BX)_{s_i} \otimes \prod_{\text{dep}(\sigma) \leq i < \text{ar}(\sigma)} X_{s_i} \right) \right) \\
[T_{e \sigma} \text{inj}_{\sigma}] &\rightarrow T_{e \sigma} \left( \Pi_{\sigma : (s_i) \rightarrow s} \left( \bigotimes_{0 \leq i < \text{dep}(\sigma)} (BX)_{s_i} \otimes \prod_{\text{dep}(\sigma) \leq i < \text{ar}(\sigma)} X_{s_i} \right) \right) \\
T_{e \sigma} &\rightarrow (T_e BX)_{s \rightarrow (T_{e \sigma} \text{inc}_X),} \rightarrow (T_e BTX)_{s

The first step discards the active arguments (via \( \pi_2 \)), which are not needed by the effect-free specification \( \rho \), as well as the behaviours of the non-active arguments (via \( \pi_1 \)). The abbreviated map \( \text{comb}(BX)_{s_1, \ldots, (BX)_{s_{\text{dep}(\sigma)}}} \) pulls out the effects of the active arguments; if there are none, the map \( \text{comb} = \eta_{\Pi_{s \rightarrow 1}} \) simply injects the empty smash product \( 1 \wedge \) into \( T_{e \sigma}(1 \wedge) \). Then the costrength \( \text{cost}(0) \otimes (BX)_{s_i} \prod_{X_{s_i}} \) attaches the non-active arguments \( \prod X_{s_i} \) to each branch of the effect-tree. The fourth map swaps the monad \( T_{e \sigma} \) and the coproduct \( \prod_{\sigma \in \text{Sig}} \), which is indexed by the set of \( s \)-sorted syntax constructors \( \{ \sigma : (s_i) \rightarrow s \in \text{Sig} \} \), as follows. For each \( s \) write \( \text{inj}_{\sigma} \) for the injection \( Y_{s \rightarrow} \rightarrow \bigotimes_{\delta \in \text{Sig}(s_i) \rightarrow s} Y_{\delta} \) given by the \( s \)-component of the coproduct (where \( Y_{s \rightarrow} = \bigotimes_{(BX)_{s_i} \prod X_{s_i}} \)). We then apply \( T_{e \sigma} \) to each injection \( \text{inj}_{\sigma} \) and take the coproduct \( [T_{e \sigma} \text{inj}_{\sigma}]_{\sigma} \) over all \( \sigma \in \text{Sig} \) to complete the swap. We then apply the map \( \rho \) defining the effect-free abstract OS specification, and the map \( \text{inc}_X : TX \rightarrow T'X \) includes terms of the effect-free language fragment \( TX \) – the initial \( (X + \Sigma) \)-algebra – among the terms \( T'X \) containing syntactic effects. We can do this as \( T'X \) is the initial \( (X + \Sigma + \Delta) \)-algebra, and it has an evident \( (X + \Sigma) \)-algebra structure; then \( \text{inc}_X \) is the initial \( (X + \Sigma) \)-algebra map \( TX \rightarrow T'X \), and it is easily shown to be natural in \( X \).

This completes our definition of \( \epsilon_X^{(2)} \), allowing us to define an abstract OS specification \( \epsilon_X : \Sigma'(X \times T_e BX) \rightarrow T_e BT'X \) for the extension of a language with syntactic effects. For instance,
given an effectless abstract OS specification \( \rho : \Sigma(X \times BX) \to BTX \) for the language SWhile along the lines of Example 3.3.17, the above definitions yield an effectful specification \( \epsilon_X : \Sigma(X \times T_eBX) \to T_eBT'X \) for the language sEWhile incorporating syntactic effects for global store.

We conclude this section by sketching how effect-trees may be introduced through custom syntax constructors \( \kappa \), rather than directly incorporating them into the language as above. For instance, instead of introducing \( \mathbb{N} \)-ary read operations \( rd_x \) into the language, one may wish to introduce (nullary) variable lookups \( x \) which behave as follows: \( x \rightarrow rd_x(0, 1, \ldots) \). (For simplicity we assume the custom commands are essentially single-sorted, of arity \( \kappa : s^a \rightarrow s \).)

This means changing the syntax functor \( \Sigma' = \Sigma + \Delta \) by replacing the effect-syntax functor \( \Delta \) with another functor of form \( (\Sigma^2X)_s = \coprod_{\kappa: s^a \rightarrow s} X_s^a \).

To define an abstract OS specification, we now require a new natural transformation \( \epsilon_X^{(3)} : \Sigma_2(X \times T_eBX) \rightarrow T_eBT'X \) which we define on each component of the above coproduct (in the \( s \)-component of \( \Sigma_2X \)).

To do this, we aim to represent the effect-trees produced by each custom command \( \kappa \) with a (single-sorted) natural transformation \( f_{\kappa X}^s : Y^a \rightarrow T_eY \) in \( \mathcal{C} \), which combines the arguments \( \tilde{y} \) of a term \( \kappa(\tilde{y}) \) into an effect-tree \( \delta(\tilde{z}) \). One then obtains the component of the map \( \epsilon_X^{(3)} \) corresponding to syntax constructor \( \kappa \) as the composition

\[
(X_s \times (T_eBX)_s)^n \xrightarrow{(\pi_1)^n} (X_s)^a \xrightarrow{f_{\kappa X}^s} T_eX_s = (T_eX)_s \xrightarrow{(T_e\text{inr}_X)_s} (T_eBX)_s \xrightarrow{(T_eB\eta_X)_s} (T_eBT'X)_s.
\]

Finally, to define the required map \( f_{\kappa X}^s \), we must inject the arguments \( X_s^a \) into \( (\Delta_0)^nX_s \) – describing effect-syntax trees \( \delta(\tilde{z}) \) of depth \( n \); and we construe these syntax-trees as arbitrary terms in \( T_eX_s \) by postcomposing with the maps \( g_{X_s}^{(n)} \), defined inductively as follows (where \( \psi_{X_s}^{\Delta_0} \) is the free \( \Delta_0 \)-algebra structure of \( T_eX_s \)).

\[
g_{X_s}^{(0)} = \eta_{X_s} : X_s \rightarrow T_eX_s \quad g_{X_s}^{(n+1)} : (\Delta_0)^{n+1}X_s \xrightarrow{\Delta_0 g_{X_s}^{(n)}} \Delta_0T_eX_s \xrightarrow{\psi_{X_s}^{\Delta_0}} T_eX_s
\]

Given an effect-free language, with syntax described by a functor \( \Sigma \), and an effect-free operational specification, regardless of whether one obtains an effectful extension through custom commands, or by introducing syntactic effects directly into the language, one arrives at a new syntax functor \( \Sigma' \) and its syntactic monad \( T' \). For the rest of this chapter, it is notationally convenient to assume such an extension has been carried out, so that we continue to use the same symbols \( \Sigma, T \) for the syntax of the extended language, without decoration.
3.4 Adequacy and Compositionality in a Mixed Kleisli Setting

3.4.1 Fine- and Coarse-grained Semantic Domains for Effects

Having obtained an operational specification for a syntactic ETS, we obtain an operational model
\[ \text{om} = T^e(\mathcal{B}_0) : T_0 \to T_eBT_0 \]
for the closed terms of the language, by structural recursion over the empty collection of generators\( \mathcal{B}_0 : 0 \to T_eB0 \). Moreover, taking base cases given by the final\( T_eB\)-coalgebra\( (D, \zeta) \) yields an operational model\( T^e(\zeta) : TD \to T_eBTD \) for terms over the final\( T_eB\)-coalgebra, which we call \text{omd}, or ‘operational model for (fine-grained) denotations’.

Then the final\( T_eB\)-coalgebra morphism\( \beta_{\text{omd}} : TD \to D \) provides an interpretation of syntax constructors on the final\( T_eB\)-coalgebra, allowing us to treat it as a fine-grained denotational model with the following Σ-algebra structure.

\[
dm : \Sigma D \xrightarrow{\Sigma \eta^T_D} \Sigma TD \xrightarrow{\psi_D} TD \xrightarrow{\beta_{\text{omd}}} D.
\]

Example 3.4.1. Consider an extension of \text{SWhile} with a syntactic binary \text{or}-effect to represent non-determinism. For each\( m \), let\( d_m \) be the denotation of a program which, for some non-deterministic choice of\( n \), returns\( m + n \) after\( n \) steps. Then\( d_0 \) would have the following transitions, exhibiting an \text{or}-effect at each step:

\[
d_0 \rightarrow 1 \text{ or } d_1 \\
\rightarrow 1 \text{ or } 2 \text{ or } d_2 \\
\rightarrow 1 \text{ or } 2 \text{ or } 3 \text{ or } d_3 \rightarrow \ldots
\]

Now consider the expression\( d_0 + 42 \). The term\( 42 \) has a trivial transition\( 42 \rightarrow 42 \); the operational specification of addition\( + \) then implies the following transitions

\[
d_0 + 42 \rightarrow +1(42) \text{ or } d_1 + 42 \\
\rightarrow 43 \text{ or } +2(42) \text{ or } d_2 + 42 \\
\rightarrow 43 \text{ or } 44 \text{ or } +3(42) \text{ or } d_3 + 42
\]

and this describes the interpretation in the denotational model\( \text{dm} \) of the\( + \) operator on\( d_0 \) and\( 42 \): it is the element of the final\( T_eB\)-coalgebra matching the above transitions, which we may represent in more detail as follows. (The occurrences of\( \text{inr}, \text{inl} \) correspond to (non)-terminal atomic transitions described by the functor\( B \); each layer of effects\( \text{or} \), written infix, corresponds to an occurrence of the functor\( T_e \); and the unit\( \eta^T_e \) is needed to describe transitions which do not introduce any effects; it is there for type-correctness.)

\[
[d_0 + 42] = \text{inr}(\eta^T_e(\text{inl}(43))) \text{ or inr}(\eta^T_e(\text{inl}(44))) \text{ or } \ldots
\]
The above construction is easily adapted to the semantic domain $T_e\overline{D}$ we have adopted, given by effect-trees $T_e$ over the initial $B$-algebra $\overline{D}$; we need to perform structural recursion with base cases given by elements of the semantic domain $T_e\overline{D}$, and this requires that we give it a $T_e B$-coalgebra structure. This is readily achieved by the following composition, which we call $\tilde{\zeta}$.

\[
\tilde{\zeta} : T_e\overline{D} \xrightarrow{T_e\alpha^{-1}} T_e B\overline{D} \xrightarrow{T_e B\eta_{\overline{D}}} T_e BT_e\overline{D}.
\]

This $T_e B$-transition structure on coarse-grained denotations implies that all the effects in a denotation are observed immediately on the first step; the individual (non-$\bot$) execution traces described by $\overline{D}$ ‘tick down’ until termination (via the map $\alpha^{-1}$) without introducing any further effects (as described by the unit $\eta_{\overline{D}}$), until a value is returned.

This time, structural recursion gives a $T_e B$-coalgebra structure to terms $T(T_e\overline{D})$, or an ‘operational model over coarse-grained denotations’ $\overline{\text{omd}}$, shown below. (As before, we use the same symbol $\overline{\text{omd}}$ for the underlying arrow $T(T_e\overline{D}) \to T_e BT(T_e\overline{D})$, and its corresponding Kleisli arrow $T(T_e\overline{D}) \to \overline{\beta}(T_e\overline{D})$.)

\[
\overline{\text{omd}} = T^\epsilon(\tilde{\zeta}) : T(T_e\overline{D}) \to T_e BT_T e\overline{D}.
\]

As $T_e B$-coalgebras coincide with $\overline{B}$-coalgebras, there is a final $\overline{B}$-coalgebra morphism $\overline{\beta}_{\overline{\text{omd}}} : T(T_e\overline{D}) \to T_e\overline{D}$ which interprets program-syntax constructors on the new semantic domain. In the same way as before, we obtain a ‘coarse-grained’ denotational model $\overline{\text{dm}}$, given by the following composition; it describes the overall effect-trees produced during executions, rather than detailing effects at every execution step.

\[
\overline{\text{dm}} : \Sigma(T_e\overline{D}) \xrightarrow{\Sigma\eta_{\overline{D}}} \Sigma T(T_e\overline{D}) \xrightarrow{\psi_{\overline{D}}} T(T_e\overline{D}) \xrightarrow{\overline{\beta}_{\overline{\text{omd}}}} T_e\overline{D}.
\]

**Example 3.4.2.** Let $\overline{d}_{n,\xi}$ be the denotation in $T_e\overline{D}$ of a program which returns $\xi$ after $n$ effect-free transition steps. For instance, the coalgebra structure $\tilde{\zeta}$ would assign to $\overline{d}_{2,\frac{1}{2}}$ the transitions $\overline{d}_{2,\frac{1}{2}} \rightarrow \overline{d}_{1,\frac{1}{2}} \rightarrow \frac{1}{2}$. Now let $d_0$ (again) be the denotation of a program which nondeterministically returns $\xi$ after $n$ steps, for some value of $n$. The map $\tilde{\zeta}$ assigns the following transition behaviour to $d_0$; note that the entire or tree is observed in the first transition, and its leaves $\overline{d}_{n,\xi}$ simply count down to termination.

\[
\begin{array}{c}
d_0 \rightarrow 1 \text{ or } \overline{d}_{1,\frac{1}{2}} \text{ or } \overline{d}_{2,\frac{1}{2}} \text{ or } \ldots \\
\rightarrow 1 \text{ or } 2 \text{ or } \overline{d}_{1,\frac{3}{2}} \text{ or } \ldots \\
\rightarrow 1 \text{ or } 2 \text{ or } \frac{3}{2} \text{ or } \ldots
\end{array}
\]

As a result, the transition behaviour of the term $d_0 + 42$ would be as follows:
\[ d_0 + 42 \rightarrow +_1(42) \quad \text{or} \quad \overline{d}_{1,2} + 42 \quad \text{or} \quad \overline{d}_{2,3} + 42 \quad \text{or} \quad \ldots \]

\[ \rightarrow 43 \quad \text{or} \quad +_2(42) \quad \text{or} \quad \overline{d}_{1,3} + 42 \quad \text{or} \quad \ldots \]

\[ \rightarrow 43 \quad \text{or} \quad 44 \quad \text{or} \quad +_3(42) \quad \text{or} \quad \ldots \]

The interpretation in \( \overline{d} \) of addition + on the denotations \( d_0, 42 \) is then given by mapping this behaviour into the final \( B \)-coalgebra: it is the denotation

\[(2, 43) \quad \text{or} \quad (3, 44) \quad \text{or} \quad (4, 45) \quad \text{or} \quad \ldots \]

describing the overall or-tree produced, along with the steps-to-termination and the return value of each computation branch.

However, in moving from a fine-grained to a coarse-grained semantic domain, one may lose compositionality and adequacy without a suitable restriction on operational specifications. We have already pointed out that abstract OS specifications are very general; the operational behaviour of syntax terms may depend on fine-grained information about which effects are observed at each execution step – and this information is not present in their coarse-grained denotations. This means that structural recursion over denotations cannot describe the behaviour of a program solely in terms of the denotations of its sub-terms; thus, structural recursion will assign a different denotation \([p]\) from the one obtained by mapping the program directly into the final \( B \)-coalgebra, \( \overline{\beta}_{\text{om}}(p) \), and we cannot have \([-] = \overline{\beta}_{\text{om}}\).

**Example 3.4.3.** Consider an interleaving operator \(|\) or a ‘one-step’ evaluator \(>:\), defined by the following effect-free rules:

\[
\begin{align*}
x & \rightarrow x' \\
x & \rightarrow y \\
x & \rightarrow x' \\
x & \rightarrow y
\end{align*}
\]

\[
\begin{align*}
x \mid y & \rightarrow y \mid x' \\
x \mid y & \rightarrow y \\
x :> y & \rightarrow y \\
x :> y & \rightarrow y
\end{align*}
\]

These rules are single-premise, permitting a natural extension to an effectful setting (where \( \text{dep}([\text{\texttt{\texttt{|}}}] = \text{dep}(>:) = 1 \)). Operationally, the effectful extension \( x \mid y \) exhibits the effects given by the first step of behaviour of \( x \), then by that of \( y \); then more effects are introduced by the successors of \( x \), then by those of \( y \), and so on. \( x :> y \) exhibits effects from the first step of \( x \)'s execution only, before evaluating \( y \).

Now consider the following two programs, \( p_1 = \text{wr}_{y,1}(\text{skip}; \text{skip}) \) and \( p_2 = \text{skip}; \text{wr}_{y,1}(\text{skip}) \). They both assign \( y = 1 \) and terminate in 3 steps; hence they both receive the denotation \( \text{wr}_{y,1}(3, \ast) \). Now suppose we put either \( p_1 \) or \( p_2 \) in the contexts \([-] \mid q \) or \([-] :> q \), where

\[ q = \text{rd}_y(\text{wr}_{x,0}(\text{skip}), \text{wr}_{x,42}(\text{skip}), \text{wr}_{x,42}(\text{skip}), \ldots) \]

then we obtain terms whose transition behaviours produce the following effect trees: The
transitions of the term \( p_1 | q \) produce the effect-tree
\[
wr_{y,1}(rd_y(wr_{z,0}(\ast), wr_{z,42}(\ast), wr_{z,42}(\ast)))
\]
Whereas \( p_2 | q \) produces the effect-tree
\[
rd_y(wr_{y,1}(wr_{z,0}(\ast)), wr_{y,1}(wr_{z,42}(\ast)), wr_{y,1}(wr_{z,42}(\ast)))
\]
Not only are these effect trees syntactically different – an issue which, in isolation, might be resolved by imposing equations on the effects – but they also correspond to different semantic behaviour: the first effect-tree corresponds to setting \( y \) to 1, looking up the value of \( y \) – which is now 1 – and thus setting \( z \) to 0. By contrast, the second tree first looks up \( y \), and then sets \( y \) to 1 and \( z \) to either 0 (if \( y \) was 0) or 42 (otherwise). Hence, even allowing for equations on the effects, the interpretations of \( p_1 | q \) and \( p_2 | q \) cannot be the same, despite the fact that \( p_1 \) and \( p_2 \) both correspond to the same element of the semantic domain. The same is true for the behaviours of \( p_1 :> q \) and \( p_2 :> q \). Thus, the denotational semantics of \( | \) or \( :> \) cannot be adequate or compositional.

To preserve compositionality, we must ensure syntax constructors only make use of information present in the denotations of their sub-terms: in other words, they cannot depend on precisely when their arguments exhibit effects, but only on the overall effect-tree produced during execution, and the number of steps-to-termination and terminal value of each computation branch. For instance, this would permit predicates which test whether any computation branch of program \( p \) terminated in \( n \) or less steps: and one might speculate that compositionality would be preserved if the ‘one step’ operator \( x :> y \) instead executed \( x \) until an effect-tree \( \delta(\tilde{b}) \) was observed, before truncating \( \delta \) at the first effect \( e \) and replacing its arguments with \( y \)’s; or similarly truncating the tree whenever a variable update \( wr_{x,42} \) is observed.

This suggests that there are many possible ways of exploiting the information present even in coarse-grained denotations. To reason about compositionality in a structured manner, we restrict attention to the kind of operational specifications considered in the previous section, where each syntax constructor \( \sigma \) has a set of active arguments, and one applies effect-free operational specifications at each branch of the effectful behaviours of those arguments. As shown by the previous example, we cannot intermingle the effects produced by different arguments; we must commit to evaluate a single active argument, until its computation branches terminate, exhibiting all the effects that appear in the denotation of that argument. In a sense, the term and the other arguments act as a ‘context’ for the evaluation of that syntax constructor, and so we will refer to it as a context-term constructor.

Examples of context-term constructors in While include addition operators \(+, +_n\), if statements, sequential composition \( ; \) and assignments \( x = u \). To evaluate them, we examine the
behaviour of a distinguished, i.e. active, argument. When it terminates with some value, we may have to evaluate another term, or produce another terminal value. (If the active argument diverges, so must the overall execution.) Some of the corresponding operational rules are shown below.

\[
\begin{align*}
(p, s) & \rightarrow (p', s') \\
(p; q, s) & \rightarrow (p', q, s') \\
(u, s) & \rightarrow (u', s') \\
(x, s) & \rightarrow (\ast, s'[x \mapsto n])
\end{align*}
\]

Alternatively, syntax constructors might behave in a way which does not depend on any behaviours of their arguments; we will call them *redex constructors*. In While, they include: elementary terms \(n \in \mathbb{N}, b \in \mathbb{B}, \text{skip}\) which immediately terminate and return \(n, b\), and \(\ast\) respectively; variable lookups \(x \in L\), returning the value \(s(x)\) of the store at \(x\); and *while* statements, which we specify with an immediate transition to an *if* statement as shown below, regardless of how the arguments behave.

\[
\langle \text{while}(e)\text{ do }\{p\}, s \rangle \rightarrow \langle \text{if}(e)\text{ then }\{p; \text{while}(e)\text{ do }\{p\}\}\text{ else }\{\text{skip}\}, s \rangle
\]

Redexes and context-terms give rise to the following congruence format, named after the concept of evaluation-in-context; see e.g. [JSV10]. This format will be the forerunner of three variants EIC1-3 on the same idea, discussed in the next chapter. We distinguish the first argument \(x_1\) of a context-term \(\sigma(x_1, \ldots)\) by writing \(\sigma(x, \bar{x})\) where the remaining arguments \(\bar{x}\) are assumed to be indexed by some set \(I\).

Note that this rule format will never derive a divergent transition \(t(\bar{x}) \rightarrow \bot\); this fact will simplify the syntactic reasoning of Theorem 3.4.7. However, to be well-defined, the format must permit the active argument \(x_1\) to diverge; in this case, we expect the term to evaluate \(x_1\) and hence diverge; so this is asserted by the format. To avoid the need for further considerations of order-structure in operational rules, we assume the collection of return values is given by a flat cppo. Moreover, when describing operational rules concretely, it is convenient to consider divergence \(\bot\) as a special return value, and we underline it: \(\bot\).

**Definition 3.4.4.** Suppose we are given an \(S\)-sorted syntax signature \(\text{Sig}\), a single-sorted effect signature \(\text{Eff}\), and a flat cppo \(\text{Vals}_\bot\) of \(S\)-sorted return values in \(\text{Cpo}^S_\bot\). An abstract OS specification \(\epsilon\) for a syntactic ETS – i.e. a \(T_B, B\)-coalgebra – is in *effectfully extended Evaluation-In-Context (eEIC) format* if: (1) it arises as an extension of a dep-supported effectless specification \(\rho'\) where \(\text{dep}(\sigma) \leq 1\) for all \(\sigma \in \text{Sig}\); and (2) the definition of \(\rho'\) amounts to a collection of operational rules, given by the following rules for each \(\sigma \in \text{Sig}\), of arity \((s_i)_{0 \leq i < \text{ar}(\sigma)} \rightarrow s\):

- Either: one of the following premise-free rules: \(\sigma(x) \rightarrow t(\bar{y})\) or \(\sigma(x) \rightarrow \nu\) – in which case \(\sigma\) is a redex constructor;
• Or: the left-most rule below, and an instance of either of the other two rules, for each value $v \in V_s$, where if $v = \bot$, then also $u = \bot$. In this case, $\sigma$ is a context-term constructor.

$$x_1 \to x_1'$$

$\sigma(x_1, \bar{x}) \to \sigma(x_1', \bar{x})$

and

$$x_1 \to v$$

$\sigma(x_1, \bar{x}) \to t(\bar{y})$

or

$$x_1 \to v$$

$\sigma(x_1, \bar{x}) \to u$

Here, we assume that $I, J$ are countable sets, $\bar{x}$ is an $I$-indexed collection of arguments, and $\bar{y}$ is a $J$-indexed collection such that $\{y_j : j \in J\} \subseteq \{x_i : i \in I\}$. (Note that the $y_j$ cannot include $x$.) In addition, $t$ is an arbitrary term with arguments given by $\bar{y}$; we require that it has the same sort $s$ as the terms containing $\sigma$, and that they are correctly typed.

Under this format, a term’s behaviour depends on at most one sub-term – without loss of generality, the first. It is executed in its place until termination, at which point the term evolves to another term depending on the final value. In the following chapter, we will express this rule format categorically; in the rest of this chapter, we reduce the problem of proving adequacy and compositionality to a large commuting diagram; and we then give a syntactic proof that the above congruence format makes it commute, by examining the transition structure it imposes on the fine- and coarse-grained operational models over denotations $\text{omd}, \overline{\text{omd}}$.

### 3.4.2 Adapting Adequacy and Compositionality from Fine- to Coarse-grained Semantics

At this stage, we review the diagram at the beginning of Section 3.2, shown again below, which displays the main ingredients of our semantic framework – although now we express the diagram entirely in the base category (whereas before it was split, with the bottom-half in the Kleisli-category).

Except for the final proof in this section, the methods of this section do not depend on a specific choice of monad $M$ in the coalgebraic behaviour functor $MB$. Hence, as we will re-use these methods in the following chapter, we generalise from the syntactic-effect monad $T_e$ to an arbitrary monad $M$ – still assuming the final $B$-coalgebra in $\text{Kl}(M)$ is given by the initial $B$-algebra. (The denotational models $\text{dm}, \overline{\text{dm}}$ can be induced in the same way as they were in Section 3.4.1, by replacing the monad $T_e$ with $M$.)

In this diagram, the dagger $\dagger$ in $(\overline{B} \beta_{\text{om}})^{\dagger}$ appears in the underlying category when we post-compose the arrow $\text{om}$ with $\overline{B} \beta_{\text{om}}$ in the Kleisli category. A similar comment applies when post-composing $\overline{\beta}_{\text{om}}$ with the final $\overline{B}$-coalgebra structure $J\alpha^{-1}$, which then requires a dagger $(J\alpha^{-1})^{\dagger}$; but we have used the fact that $(J\alpha^{-1})^{\dagger} = M\alpha^{-1}$, by Remark 2.2.4.
Given any abstract operational specification \( \epsilon_X : \Sigma(X \times MBX) \to MBTX \), we can obtain an operational model \( \text{om} \), and both fine- and coarse-grained denotational models \((D, \text{dm}), (\overline{M}D, \overline{\text{dm}})\) respectively as in Section 3.4.1. Operational equivalence of programs is described by the final \( \overline{B} \)-coalgebra morphism \( \overline{\beta}_\text{om} \) from the operational model into the final \( \overline{B} \)-coalgebra \( \overline{D} \); by contrast, the assignment \([-]\) of denotations to programs is given by the initial \( \Sigma \)-algebra morphism from \( T0 \) into the denotational model. To prove adequacy and compositionality for our semantics, we must show that the central arrows coincide; we will do this by showing that the final \( \overline{B} \)-coalgebra morphism \( \overline{\beta}_\text{om} : T0 \to \overline{D} \) is, like \([-]\), a \( \Sigma \)-algebra morphism from \( T0 \) into the denotational model \((\overline{M}D, \overline{\text{dm}}) : \Sigma\overline{M}D \to \overline{M}D\). As \( T0 \) is the initial \( \Sigma \)-algebra, there can be only one such \( \Sigma \)-algebra morphism, and this will imply \([-] = \overline{\beta}_\text{om} \).

The original proof by Turi and Plotkin, in terms of bialgebras, may be applied to the fine-grained semantic domain given by the final \( MB \)-coalgebra \( D \). To do this, we re-use our operational model \( \text{om} \) for effectful programs, and the denotational model \( \text{dm} \), to obtain initial \( \Sigma \)-algebra and final \( MB \)-coalgebra morphisms \([-], \beta_\text{om} \) into \( D \), which are shown to be equal by the argument in Section 2.1.9. This equality is expressed in the below diagram.

Unfortunately, there is no obvious way of adapting the bialgebraic proof of Turi and Plotkin to a mixed Kleisli setting. One quickly finds that there is no natural candidate for a lifting of the program-syntax monad \( T \) to the category of \( \overline{B} \)-coalgebras; in the underlying category, this would require a means of producing arrows \( TX \to MTY \) from \( \overline{B} \)-coalgebra morphisms \( X \to MY \). There are similar difficulties in attempting to define a suitable cofree comonad. One may argue that a simple adaptation of the original proof is unlikely to work, because the original form of operational specifications \( \epsilon \) must be restricted somehow – for instance by the evaluation-in-context format – for compositionality and adequacy to hold; and due to the generality of possible restrictions discussed after Example 3.4.3, there is unlikely to be a
canonical choice. This is why we have taken a less abstract route, and given a direct proof that the final $B$-coalgebra morphism $\beta_{om}$ is also a $\Sigma$-algebra morphism.

To do this, we aim to exploit the adequacy and compositionality of the fine-grained semantics in the previous diagram. It is straightforward to map fine-grained denotations $D$ into coarse-grained denotations $MD$, as the final $MB$-coalgebra is also a $B$-coalgebra, hence there is a unique $B$-coalgebra morphism $\beta_{om}$ into $D$, of underlying type $D \to MD$. (As we did for the operational model $om$, for convenience we have used the same symbol $\zeta$ for the coalgebra structure of $D$ in the Kleisli category, and for its underlying arrow.)

One may interpret the map $\beta_{om}$ as ‘coarsening’ the fine-grained denotations by forgetting information about which effects are observed at each execution step, leaving only the overall effect-trees produced by denotations (and the pairs $(n, v) \in D$ of the steps-to-termination and return value at each computation path). This allows us to extend the above diagram as shown below: a square is added on the bottom-right expressing the fact (in the underlying category) that $\beta_{om}$ is a $B$-coalgebra morphism.

As one might expect, the final $B$-coalgebra morphism $\beta_{om} : T0 \to MD$ – assigning coarse-grained denotations to programs – may be shown to factorise as shown above, through the assignment of fine-grained denotations $\beta_{om} : T0 \to D$ and the coarsening map $\beta_{z}$ : $D \to MD$, by the following reasoning. First, one may easily show that if $g : X \to Y$ is an $MB$-coalgebra morphism between two $MB$-coalgebras $(X, \gamma), (Y, \delta)$, then $Jg$ is a $B$-coalgebra morphism between the same coalgebras in the Kleisli category, construed as $B$-coalgebras. Thus, the $MB$-coalgebra morphism $\beta_{om} : T0 \to D$ gives rise to a $B$-coalgebra morphism $J\beta_{om} : T0 \to D$; hence we may post-compose (in the Kleisli category) with the $B$-coalgebra morphism $\beta_{z}$ to obtain another $B$-coalgebra morphism $\beta_{z} \circ J\beta_{om} : T0 \to D$. However, by finality there is only one $B$-coalgebra morphism from the operational model $T0$ into $D$; hence $\beta_{om} = \beta_{z} \circ J\beta_{om}$ in the Kleisli category. Finally, by Remark 2.2.4 again, pre-composition with an arrow $Jg$ in the Kleisli-category is equivalent to pre-composing with $g$ in the underlying category, hence $\beta_{om} = \beta_{z} \circ \beta_{om}$.

Using this factorisation of $\beta_{om}$, to prove adequacy and compositionality it is sufficient to prove that the coarsening map $\beta_{z}$ is a $\Sigma$-algebra morphism from $(D, dm)$ to $(D, dm)$: this is the top-right square (*) in the diagram below. Turi and Plotkin’s result (applied to $MB$-coalgebras)
has already told us that \( \beta_{\text{om}} \) is a \( \Sigma \)-algebra morphism; hence the composition \( \overline{\beta}_{\zeta} \circ \beta_{\text{om}} = \overline{\beta}_{\text{om}} \) must also be a \( \Sigma \)-algebra, as required.

\[
\begin{array}{ccc}
\Sigma T 0 \xrightarrow{\Sigma \beta_{\text{om}}} & \Sigma D \xrightarrow{\Sigma \overline{\beta}_{\zeta}} & \Sigma M \overline{D} \\
\downarrow \psi_0 & \downarrow \delta m (\ast) & \downarrow \delta m \\
T 0 \xrightarrow{\beta_{\text{om}}} & D \xrightarrow{\overline{\beta}_{\zeta}} & M \overline{D} \\
\downarrow \text{om} & \downarrow \beta_{\text{om}} & \downarrow \text{omd} \\
MBT 0 \xrightarrow{(B(\overline{\beta}_{\zeta} \circ \beta_{\text{om}}))^{-1}} & MB \overline{D}
\end{array}
\]

It will be convenient to generalise the single layer of syntax described by \( \Sigma \) in condition (\( \ast \)) to arbitrary-depth syntax given by \( T \)'s: this may be done as follows, giving the square (\( + \)) below. (The composition of the upper three horizontal arrows is \( \delta m \), and the bottom arrows are \( \overline{\delta m} \).)

\[
\begin{array}{ccc}
\Sigma D \xrightarrow{\Sigma \eta T} & \Sigma TD \xrightarrow{\psi D} & TD \xrightarrow{\beta_{\text{omd}}} & D \\
\downarrow \Sigma \overline{\beta}_{\zeta} & \downarrow \Sigma T \overline{\beta}_{\zeta} & \downarrow T \overline{\beta}_{\zeta} (\ast) & \downarrow \overline{\beta}_{\zeta} \\
\Sigma M \overline{D} \xrightarrow{\Sigma \eta T} & \Sigma TM \overline{D} \xrightarrow{\psi M \overline{\eta}} & TM \overline{D} \xrightarrow{\beta_{\text{omd}}} & M \overline{D}
\end{array}
\]

The first square commutes as it is the image under \( \Sigma \) of the naturality of \( \eta T \). The second commutes because \( \psi_X : \Sigma TX \to TX \), the \( \Sigma \)-algebra structure of \( TX \), is a natural transformation. This is an easy consequence of the definition of \( T \) by adjunction, \( T = U^\Sigma P^\Sigma \) (Section 2.1.3).

One may interpret the condition (\( + \)) as an abstract constraint on the abstract OS \( \epsilon \) as follows. \( TD \) describes terms over the coarse-grained denotational model \( D \), whose arguments introduce effects at each transition step. The upper path assigns an overall effect-tree to these terms, based on their behaviour specified by \( \epsilon \). The lower path (via \( T \overline{\beta}_{\zeta} \)) collects the effects of each argument into its \textit{first} execution step, giving an element of \( TM \overline{D} \), and then similarly assigns an overall effect-tree. Thus, (\( + \)) asserts that the execution step at which effects occur (in the arguments \( D \) of terms \( TD \)) is irrelevant, as they might as well all be in the first step.

### 3.4.3 Reducing Adequacy and Compositionality to a Condition of Cones

We will characterise the condition (\( + \)) more concretely, in terms of cones in the Kleisli category. Recall that by algebraic \( \omega \)-compactness of \( \text{Cpo}_{\perp} \), the initial \( B \)-sequence converges after \( \omega \) steps; the proof of Theorem 2.2.16 showed that the same was true for the initial \( \overline{B} \)-sequence in \( \text{Kl}(M) \). (Also recall that left-strictness implies that \( M0 = 1 \), so that 0 is the final object in \( \text{Kl}(M) \).)
**Definition 3.4.5.** The cone (over the final sequence up to $\omega$) generated by a $\overline{B}$-coalgebra $(X, \gamma)$ consists of the arrows $(\gamma_n : X \to \overline{B}^n 0)_{n<\omega}$ obtainable by composition in the following diagram:

$$
\begin{array}{cccccccc}
X & \xrightarrow{\gamma} & \overline{B}X & \xrightarrow{\overline{B}\gamma} & \overline{B}^2X & \xrightarrow{\overline{B}^2\gamma} & \cdots \\
\downarrow{}^1x & && \downarrow{}^{\overline{B}x} & && \downarrow{}^{\overline{B}^2x} & \\
0 & \xleftarrow{0} & \overline{B}0 & \xleftarrow{\overline{B}0} & \overline{B}^20 & \xleftarrow{\overline{B}^20} & \cdots \\
\end{array}
$$

The limit-colimit coincidence of Theorem 2.2.16 implies that $((J\alpha^{-1})_n : \overline{D} \to \overline{B}^n 0)_{n<\omega}$ is a limiting cone. It is straightforward to show:

**Lemma 3.4.6.** On a category $\mathcal{C}$, suppose a final $H$-coalgebra $(\overline{D}, \zeta)$ exists, and is the limit of the final $H$-sequence up to ordinal $\omega$. Then the final $H$-coalgebra morphism from any coalgebra $(X, \gamma)$ into $(\overline{D}, \zeta)$ must coincide with the mediating morphism from the cone $(\gamma_n)$ generated by $(X, \gamma)$ into the limiting cone $(\zeta_n)$.

We will show that we can describe the paths in condition $(+)$, shown again below, as mediating morphisms from the vertex $TD$ of two cones $(X_n), (Y_n)$ over the final $\overline{B}$-sequence into the vertex $M\overline{D}$ of the limiting cone $((J\alpha^{-1})_n)$. We will show that the mediating morphisms are the same, by proving that the two cones are the same, i.e. that $X_n = Y_n$ for all $n$; there can only be one mediating morphism from a cone into the limiting cone.

$$
\begin{array}{ccc}
TD & \xrightarrow{\beta_{\text{omd}}} & D \\
\downarrow{}^{\overline{\beta}_\zeta} & (+) & \downarrow{}^{\overline{\beta}_\zeta} \\
TM\overline{D} & \xrightarrow{\beta_{\text{omd}}} & M\overline{D} \\
\end{array}
$$

We now describe these cones in more detail. The path $\overline{\beta}_\zeta \circ \beta_{\text{omd}}$ is a composition of $\overline{B}$-coalgebra morphisms, so it is also one, and hence by Lemma 3.4.6 must coincide with the mediating morphism from $(\text{omd}_n)$ into the limiting cone $((J\alpha^{-1})_n)$. In the other path, $\beta_{\text{omd}}$ coincides with the mediating morphism from $(\text{omd}_n)$ to $((J\alpha^{-1})_n)$ by the same Lemma, so precomposing with $T\overline{\beta}_\zeta$ gives another cone $(\text{omd}_n \circ T\overline{\beta}_\zeta)$ over the final $\overline{B}$-sequence, this time with mediating morphism $\beta_{\text{omd}} \circ T\overline{\beta}_\zeta$ into the limiting cone $((J\alpha^{-1})_n)$.

We thus aim to show that both cones over the final sequence are the same – i.e., that $\text{omd}_n = \overline{\text{omd}}_n \circ T\overline{\beta}_\zeta$. The situation is depicted below; we must show that both paths $TD \to \overline{B}^n 0$ coincide for all $n$. (The top-right square commutes as 0 is final in $\text{Kl}(M)$; by applying $\overline{B}^n$, this implies the whole right-hand column commutes.)
3.4. Adequacy and Compositionality in a Mixed Kleisli Setting

3.4.4 A Syntactic Proof of the Condition of Cones, Under the eEIC Format

Now we will prove by a syntactic argument that the previous diagram commutes in the case where \( M = T_e \), under the assumption that the abstract OS \( \epsilon \) is in eEIC format. The rest of this chapter is devoted to proving the following theorem.

**Theorem 3.4.7.** Let \( M = T_e \), \( BX = \text{Vals}_\bot \oplus X \), and \( \epsilon \) be a specification in eEIC format for a language given by extending an effectless specification \( \rho \) (either directly with syntactic effects, or the custom commands of Section 3.3.4.) Then \( \text{omd}_n = \text{omd}_n \circ T\beta_\zeta \). Thus condition (+) holds, and the denotational and operational maps \([\cdot], \beta_\text{om} \) induced by the initial and final (co)algebra morphisms \( T0 \to T_e\bar{D} \) coincide.

Left-strictness plays an important part in the proof, asserting that any all-\( \bot \) subtrees \( \delta((\bot)_{i \in I}) \) of an effect-tree may be equivalently replaced by \( \bot \).

We will make reference to two ‘horizontal’ paths in the above diagram: the ‘upper path’ (up to \( n \) steps) is the following sequence of arrows, describing successive transitions of the operational model over fine-grained denotations.

\[
TD \xrightarrow{\text{omd}} BTD \xrightarrow{\beta_\text{omd}} B^2TD \xrightarrow{\beta_\text{omd}} \cdots \xrightarrow{\beta_\text{omd}} B^nTD.
\]

The ‘lower path’ is the similar sequence which first applies \( T\beta_\zeta \), ‘coarsening’ the denotations \( D \) inside terms, and then produces the transitions of the operational model over coarse-grained denotations.

\[
TD \xrightarrow{T\beta_\zeta} T(T_e\bar{D}) \xrightarrow{\text{omd}} B(T_e\bar{D}) \xrightarrow{\beta_\text{omd}} B^2(T_e\bar{D}) \xrightarrow{\beta_\text{omd}} \cdots \xrightarrow{\beta_\text{omd}} B^nT(T_e\bar{D}).
\]

The endpoints of these horizontal arrows in the underlying category, respectively \( T_eB^nTD \) and \( T_eB^nT(T_e\bar{D}) \), describe effect-trees with leaves given by \( n \)-step execution traces (for terms in \( TX \), where \( X = D \) or \( X = T_e\bar{D} \) respectively). Each trace may be either: a completed trace \( \text{inr}(\text{inr}(\cdots(\text{inl}(v)) \cdots)) \) terminating with a value \( v \) in \( \text{Vals}_\bot \) in \( n \) steps or fewer; or it may be
an incomplete trace \( \text{inr}(\cdots (\text{inr}(t)) \cdots) \) ending with a term \( TX \) over denotations in exactly \( n \) steps. When the horizontal arrows are composed with the vertical arrows \( \overline{B}^n \text{!}_{TD} : \overline{B}^n D \to \overline{B}^n 0 \) and \( \overline{B}^n \text{T}(T_\epsilon D) : \overline{B}^n (T_\epsilon D) \to \overline{B}^n 0 \) respectively, any non-terminal leaves \( t \) in the effect-trees are replaced with \( \bot \).

Thus, for any \( n \), to show the two paths \( TD \to \overline{B}^n 0 \) agree, it is enough to show the following claim: when applied to any term \( t(d_1, \ldots) \in TD \), the horizontal paths to \( \overline{B}^n TD \) and \( \overline{B}^n (T_\epsilon D) \) produce effect-trees which agree up to occurrences of terminal values \( \underline{v} \) – and may differ in other arguments.

To illustrate this ‘agreement’ more concretely, consider the following pair of distinct effect-trees, where \( t_i \) are arbitrary terms \( TD \) over fine-grained denotations \( D \), and \( \overline{t}_i \) are arbitrary terms \( T(T_\epsilon D) \) over coarse-grained denotations \( T_\epsilon D \).

\[
\text{inr}(t_1) \quad \text{or} \quad \text{inr}((\text{inl}(42) \quad \text{or} \quad \text{inr}(t_2)) \quad \text{and} \quad \text{inr}(\overline{t}_3) \quad \text{or} \quad \text{inr}(\text{inl}(42) \quad \text{or} \quad \text{inr}(\overline{t}_4) \quad \text{or} \quad \bot).
\]

Any non-terminal leaves \( t_i \) will be mapped to \( \bot \) by the vertical arrows \( \overline{B}^n \text{!}_{TD} \) and \( \overline{B}^n \text{T}(T_\epsilon D) \) – applying these maps gives the following trees.

\[
\text{inr}(\bot) \quad \text{or} \quad \text{inr}((\text{inl}(42) \quad \text{or} \quad \bot)) \quad \text{and} \quad \text{inr}(\bot) \quad \text{or} \quad \text{inr}(42) \quad \text{or} \quad (\bot \quad \text{or} \quad \bot).
\]

One identifies \( \text{inr}(\bot) \) with \( \bot \), by definition of the coalesced sum \( \oplus \) in \( \text{Cpo}_\Delta \), giving:

\[
\bot \quad \text{or} \quad \text{inr}(42) \quad \text{or} \quad \bot) \quad \text{and} \quad \bot \quad \text{or} \quad \text{inr}(42) \quad (\bot \quad \text{or} \quad \bot).
\]

Due to the strictness of the effect-syntax functor \( \Delta \) (from its definition in terms of smash and Cartesian products), the functor \( T_\epsilon \) is also strict, by Proposition 3.3.11. Concretely, this means that any all-\( \bot \) subtrees of terms in \( T_\epsilon \) (such as \( (\bot \quad \text{or} \quad \bot) \) above) are equivalent to \( \bot \); hence the above two effect-trees agree up to occurrences of terminal values.

As we did in Definition 3.4.4 for eEIC specifications, we treat divergence as a special return value \( t(\tilde{x}) \to \bot \) – this conveniently allows us to describe both divergence and termination by the notation \( v \).

We now prove the claim by strong induction on \( n \). The \( n = 0 \) case is immediate, if we take the upper horizontal path to be the identity arrow \( \text{id}_{TD} \) in \( \text{Kl}(M) \), given by \( \eta_{T_\epsilon} \) in the underlying category. This maps any term \( t \) (over fine-grained denotations) to the effect-tree \( \eta_{T_\epsilon}(t) \) consisting of a single leaf \( t \). The lower horizontal path, via \( T\overline{B}_\zeta \), first maps \( t \) to a term \( t' \) over coarse-grained denotations, and then the identity arrow \( \text{id}_{T(T_\epsilon D)} \) also sends this to a singleton effect-tree \( \eta_{T_\epsilon}(t') \); so the two paths trivially produce effect-trees whose leaves agree up to terminal values, because there are none.
Now we assume the inductive hypothesis for all \( m < n \). The action of both horizontal paths consist of iterating the \( B \)-coalgebra maps \( \text{omd} \) and \( \text{omd} \); and we have assumed these are induced by effect-free derivation rules \( \rho \), extended into an effectful specification \( \epsilon \).

The remainder of this proof is largely concerned with the relationship between the effectful behaviour of terms \( TD \) along the upper horizontal path, and the behaviour of the corresponding term \( T(T_eD) \) along the lower path. Once both behaviours are described in detail, it becomes relatively straightforward to apply induction. The difficulty is that in the lower path, after coarsening the fine-grained denotations \( d \) in terms \( t(\tilde{d}) \in TD \), one obtains a term over coarse-grained denotations \( t'(\tilde{d}') \in T(T_eD) \) (where each leaf coarse-grained leaf \( d' \) is given by coarsening one of the fine-grained leaves \( d \)). The transition structure \( \tilde{\zeta} \) for coarse-grained denotations produces the overall effect-trees of the coarsened leaves \( T_eD \) execution in a single transition step, as illustrated by Example 3.4.2; hence, at each transition step the lower path introduces effects from arbitrarily deep within the original, fine-grained leaves \( d \). By contrast, in the upper horizontal path, one sees the gradual introduction of effects by terms \( TD \) over fine-grained denotations, as seen in Example 3.4.1.

Although this means that the lower horizontal paths may exhibit more effects than the upper paths, when both paths are post-composed with the vertical arrows \( B^n!_{TD} \) and \( B^n!_{T(T_eD)} \), the leaves of these ‘extra’ effects will be replaced with \( \perp \), as described earlier; and by strictness of the monad \( T_e \), or equivalently the fact that \( e(\perp, \ldots, \perp) = \perp \) for all effects \( e \), we will obtain the same effect tree. Conceptually, this is why strictness of the effects, and hence left-strictness of Kleisli composition, is essential for our proof: it allows us to make a correspondence between the horizontal paths of length \( n \). This idea is also central to our proof of the categorical version of this theorem in the next chapter.

**Deriving Transition Behaviours Under the eEIC Format**

We consider how the behaviours of terms over denotations \( TD, T(T_eD) \) are derived from the operational rules of specifications \( \epsilon \) in the eEIC format: these derivation rules take one of two forms. The first is for syntax constructors \( \sigma(\tilde{x}) \) which introduce effects – given either by the effect-syntax constructors added directly to the language, or through custom commands. In our definition of effectful extension (more specifically the natural transformations \( \epsilon^{(1)} \) or \( \epsilon^{(3)} \)), we described such terms by redexes, given syntactically by premise-free rules, which may introduce effect-trees of behaviours, and where each branch has a non-terminal transition to some state \( x_i \). An example would be

\[
\text{rd}_u(\tilde{x}) \rightarrow \text{rd}_u(\tilde{x}).
\]

One also has rules for the constructors \( \sigma(\tilde{x}) \) given by the effect-free fragment of the language. These rules also come in two varieties: single-premise (for context-terms), where the behaviour
Chapter 3. MOS in a Mixed Kleisli Category for Syntactic Effects

of the first argument is relevant (i.e. \( \text{dep}(\sigma) = 1 \)), such as most of the operators of \( \text{seWhile} \); and premise-free rules (for redxes), where \( \text{dep}(\sigma) = 0 \) – such as \( \text{while} \) statements and constants.

Now we consider the kind of rule derivations used to deduce the behaviour of arbitrary terms \( t(\tilde{x}) \) over variables \( \tilde{x} \). Due to our restrictions on the use of products in syntax functors in \( \mathbb{Cpo}_{\bot} \), these terms may be infinite-depth; however, we will see that rule derivations under the eEIC format will always be finite depth, due to the following observation. For a signature \( \text{Sig} \) and dependency function \( \text{dep} \), the syntax functor \( \Sigma \) represents context-term constructors \( \sigma : (s_i) \rightarrow s_{\sigma} \) (where \( \text{dep}(\sigma) = 1 \)) by a product \( X_{s_0} \otimes \prod_{1 \leq i < \text{ar}(\sigma)} X_{s_i} \). Syntactically, this smash product with the first (active) argument \( X_{s_0} \), means that one may not have a term given by an infinite series of context-term constructors \( t = \sigma_1(\sigma_2(\ldots, t_2), t_1) \) nested in the first arguments; instead, the series must end with either a variable \( x \), viz.

\[
\sigma_1(\sigma_2(\cdots(\sigma_n(x, \tilde{t}_n), \ldots, \tilde{t}_2), \tilde{t}_1)
\]

or similarly a redex-term \( t_{n+1} \) in place of \( x \). By contrast, all other sub-terms in \( \tilde{t}_n \) of the syntax constructors \( \sigma_i \) may be infinite-depth, and so may the arguments of redex terms \( \rho(\tilde{t}) \).

Under the eEIC format, the behaviour of the term \( t \) shown above may only depend on the behaviour of the innermost first argument \( x \) or \( t_n \), and the syntax constructors \( (\sigma_i)_{i \leq n} \) nested within the first positions of the constructors used to build \( t \). It will be convenient to consider the inner-most first argument \( x \) of such a term \( t \) as the ‘first’ argument of the term \( t \); suitably indexing the other arguments allows us to write the term as \( t(x, \tilde{x}) \).

Now consider an arbitrary term \( t = \sigma_1(\sigma_2(\ldots, \tilde{s}_2), \tilde{s}_1) \) over variables \( \tilde{x} \), where we make no assumption on the series of syntax constructors \( \sigma_n \) nested in first position. They may end with a nullary syntax constructor (necessarily a redex constructor, as context terms have at least one argument) or a denotation \( d \); or they may continue to infinite depth. If any of the above constructors \( \sigma_i \) is a redex, defined by premise-free derivation rules, the behaviours of its arguments are irrelevant. Hence, the derivation of the behaviour of the term \( t \) will begin with the first redex constructor \( \sigma_j \). For instance, the first redex in the nested first arguments of

\[
((\text{wr}_{x,42}(\text{wr}_{y,42}(p))) + 5) \ast 2
\]

is \( \text{wr}_{x,42}(\text{wr}_{y,42}(p)) \), so \( j = 3 \); and the derivation of its behaviour will start with the premise-free rule for that redex term:

\[
\text{wr}_{x,42}(\text{wr}_{y,42}(p)) \rightarrow \text{wr}_{x,42}(\text{wr}_{y,42}(p))
\]

Note that by our previous observation, the syntax terms cannot contain an infinite series of context terms; hence if they do not contain a redex constructor, they must eventually end with a variable \( x \) (in which case we define \( j = n + 1 \)). For example, let \( n, m \) be syntax variables; the
term \((n \ast m) + \text{wr}_{x,5}(p)\) has a sequence of first-position arguments given by \((n \ast m)\) and then \(n\) – of length 2. Neither is a redex; so we take \(j = 2 + 1 = 3\), and the derivation of its behaviour will begin with the behaviour of the inner-most first argument \(n\), e.g.

\[
\frac{n \rightarrow 5}{n \ast m \rightarrow *_5(m)} \quad \frac{n \ast m \rightarrow *_5(m) + \text{wr}_{x,5}(p)}{}
\]

In general, the derivation of the behaviour of a term will begin with the premise-free rule of the first redex-term constructor \(\sigma_j\), or the behaviour of the inner-most first argument \(x\) (where \(j\) is then defined as above); the rest of the derivation will depend on the context-term constructors \(\sigma_i\) for \(i < j\), which come from the effect-free fragment of the language. The effectful extension determines the behaviour of a context-term \(\sigma_i(x_i, \tilde{x}_i)\) in terms of the behaviour of the first argument \(x_i \rightarrow \delta(\tilde{y})\), as follows: it applies the (effect-free) rule for \(\sigma_i\) at each branch of the behaviour of \(x_i\), i.e. with the premise \(x_i \rightarrow y_k\) for each branch \(y_k\) of the effect tree \(\delta(\tilde{y})\). Similarly, for \(i < j\), none of the context-term constructors \(\sigma_i\) will introduce any further effects; hence the derivation of the behaviour of \(t\) continues at each leaf of the same effect-tree \(\delta\).

To illustrate this process, consider extending \textbf{SWhile} with a binary \textbf{or} effect, and suppose that one has a transition \(p \rightarrow q\) or \(\overline{5}\). Then, considering the behaviour of \(+_7(p)\) given by the operational rules of \textbf{SWhile}, one has the following effect-free derivations ‘at each branch of the behaviour of \(p\):

\[
\frac{p \rightarrow q}{+_7(p) \rightarrow +_7(q)} \quad \frac{p \rightarrow 5}{+_7(p) \rightarrow 12}
\]

and hence, one derives the behaviour \(+_7(p) \rightarrow +_7(q)\) or \(12\); by construction, the shape of this effect-tree \((\cdot)\) or \((\cdot)\) is the same as the effect-tree of the behaviour of \(p\) (viz. \(q\) or \(\overline{5}\)). Now consider a further layer of syntax \((+_7(p)) * 3\); this time, we apply the following effect-free derivations at each leaf of the behaviour of \(+_7(p)\):

\[
\frac{+_7(p) \rightarrow +_7(q)}{+_7(p) \ast 3 \rightarrow +_7(q) \ast 3} \quad \frac{+_7(p) \rightarrow 12}{(+_7(p)) \ast 3 \rightarrow *_{12}(3)}
\]

This derives the behaviour \(+_7(p) \ast 3 \rightarrow +_7(q) \ast 3\) or \(*_{3}(12)\), once again containing the same shape of effect tree \((\cdot)\) or \((\cdot)\) as the behaviour of the inner-most first argument \(p\).

We may combine each of the above two derivation steps, and consider the derivation of the behaviour of \(+_7(p) \ast 3\) to consist of two nested effectless derivations, one at each leaf of the effect tree \((\cdot)\) or \((\cdot)\) exhibited by \(p\):

\[
\frac{p \rightarrow q}{+_7(p) \rightarrow +_7(q)} \quad \frac{p \rightarrow 5}{+_7(p) \rightarrow 12} \quad \frac{+_7(p) \rightarrow 12}{+_7(p) \ast 3 \rightarrow *_{3}(12)}
\]
Returning to the possibility of a term $t$ containing a redex term $t_r(\tilde{x}_r)$ in the nested first position (at depth $j$), the above process gives rise to a similar effect-tree of effectless derivation rules, this time starting with the behaviour of that redex term. Suppose it is $t_r(\tilde{x}_r) \to \delta((t_i(\tilde{x}_i))_{i \in I})$. The effectless derivation at the $i$th leaf in the effect-tree $\delta$ starts with a premise $t_r(\tilde{x}_r) \to t_i(\tilde{x}_i)$.

For instance, to derive behaviour for the term $t = (\mathsf{wr}_{x,42}(p) + 5) * 2$, one starts with the premise 

$$\mathsf{wr}_{x,42}(p) \to \mathsf{wr}_{x,42}(p)$$

which introduces an effect-tree $\mathsf{wr}_{x,42}(\cdot)$ with a single branch, $p$. One then has a series of effect-free derivations at that branch, as shown. (We use an overline for the first step of the derivation at each branch of the effect-tree, to record that the first step came from a premise-free redex rule.)

$$\overline{\mathsf{wr}_{x,42}(p) \to p}$$
$$\overline{\mathsf{wr}_{x,42}(p) + 5 \to p + 5}$$
$$\overline{(\mathsf{wr}_{x,42}(p) + 5) * 2 \to (p + 5) * 2}$$

Putting the derived behaviour back in the context $\mathsf{wr}_{x,42}(\cdot)$ gives us the required behaviour, 

$$(\mathsf{wr}_{x,42}(p) + 5) * 2 \to \mathsf{wr}_{x,42}((p + 5) * 2).$$

This reasoning allows us to describe the behaviour-derivation of a term $t(\tilde{x})$ as an effect-tree of effectless derivations, each of which falls into one of the following four types. We have introduced a convention for later convenience: whenever an argument, term, or denotation would appear after a transition $\to$, we decorate it with an additional $'$ . We write $.(^{(m)})$ for $m$ primes $'\ldots'$.  

$$\overline{x_1 \to x'_1}$$
$$\vdots$$
$$t(x_1, \tilde{x}) \to t(x'_1, \tilde{x})$$

$$\overline{t_r(\tilde{x}_r) \to t_i(\tilde{x}_i)}$$
$$\vdots$$
$$t(\tilde{x}) \to t'(\tilde{x'}) \text{ or } \overline{u}$$

$$\overline{x_1 \to \overline{u}}$$
$$\vdots$$
$$t(x_1, \tilde{x}) \to \overline{u}$$

Under the eEIC format, the top-left form is the only sort of derivation possible from a premise $x_1 \to x'_1$; all other arguments $\tilde{x}$ of $t$ are fixed throughout. If the premise is $x_1 \to \overline{v}$ for some $\overline{v}$, the rule conclusions may either indicate termination (bottom-left) or give rise to a new term with a (possibly new) first argument, $t'(\tilde{x'})$ (bottom-right). The top-right case will occur if (and only if) there is a redex term $t_r(\tilde{x}_r)$ in nested first position in $t$; whatever effect tree $\delta$ it produces in its transition $t_r(\tilde{x}_r) \to \delta((t_i(\tilde{x}_i))_{i \in I})$ will also be exhibited by the term $t(\tilde{x})$, and
δ will not depend on any of the arguments ˜x (or ˜xr), but only on the redex constructor in tr. In contrast, the other three cases will introduce effects from the behaviour of the inner-most first-position argument x of t(x, ˜x), if there are no redex terms σi in first position. Note also that if an argument x1 diverges directly, the format implies that the derivation must take the bottom-left form, where every step of the derivation involves a divergent transition, including the conclusion.

From Single to Multiple Transitions in the Upper-horizontal Path

The deductions described above define the transitions t(˜d) → δ(˜b) of terms over denotations TD and T(TeD) (where the arguments ˜b are elements of BTD and BT(TeD) respectively.) We consider how these deductions are related to the derivations of successive transitions, t → δ(˜b) → δ((εi(˜bi))i∈I) → etc., in terms of the denotations at the leaves of t. To do this, we may represent d1 graphically as follows, following the diagrams introduced in Section 3.3.2:

Triangle m1 represents the effectful behaviour of d1 ∈ D under the final-coalgebra structure ζ; one may write d1 → m1((b′i)i∈I), where either b′i = v′i or b′i = (d′1)i, and we have essentially omitted the subscript i above. Thus, the leaves b′i, elements of BD, are either elements (d′1)i of D – collectively represented above by the label d′1 – or terminal values similarly represented by v′i (which may be ⊥′, if explicit divergence occurs). Likewise, the triangle m2 represents a collection of effectful behaviours of each leaf d′1, and so on. (Each leaf d′1 has its own effect-tree (m2)d′1, but to keep diagrams simple, we omit these subscripts too.) Note that the triangles mi may be degenerate, corresponding to effect-free transitions; in this case, they contain only a single leaf and no effects (essentially given by the monad unit ηTe) – either another denotation d(α), or a terminal value v.

Now we consider the effectful behaviour of terms in TD, in terms of the arguments d. We first focus on the case where the term t(d1, ˜d) has no redex constructors in nested first position; hence its immediate transition behaviour will exhibit the same effect-tree m1 as the inner-most first argument d1. However, the leaves will be different: for each leaf d′1 of the effect-tree m1 in d1, there is a corresponding leaf t(d′1, ˜d) derived according to the top-left case above. If a leaf of d1 is given by a terminal value v′, the corresponding leaf in the behaviour of t(d1, ˜d) will instead be a terminal value u′ (the bottom-left case) or an altogether new term t′(e′) (the bottom-right case).

This transition behaviour of the term t(d1, ˜d) – to values v′, or successor terms t(d′1, ˜d) and
\(t'(\tilde{e}')\) – is described by the first arrow \(TD \xrightarrow{\omega} \overline{B}TD\) in the upper horizontal path; the ensuing arrows \(\overline{B}^{n-1} \xrightarrow{\omega} \overline{B}^{n}TD\) describe \((n - 1)\) further transitions of the successor terms. We will focus on the successive behaviours of the leaves \(t(d'_1, \tilde{d})\), as they will be more prominent in this proof than the other leaves \(t'(\tilde{e}')\) or \(\tilde{e}'\).

In the first transition, an effect-tree of transitions \(t(d_1, \tilde{d}) \rightarrow m_1(\tilde{b})\) is introduced (as shown in Diagram 3.1); recall that each leaf of this behaviour is given by an effectless derivation of the statement \(t(d_1, \tilde{d}) \rightarrow b_i\). In a similar way, if \(b_i\) is a term \(t(d'_1, \tilde{d})\) (involving a ‘successor’ \(d'_1\) of \(d_1\), rather than a terminal value), then successive transitions \(t(d'_1, \tilde{d}) \rightarrow \delta'(\cdots)\) are given by some set of effectless derivations \(t(d'_1, \tilde{d}) \rightarrow b'_j\).

We may collect these transitions over \(m\) successive steps together, \(t(d_1, \tilde{d}) \rightarrow \ldots \rightarrow t(d_{1}^{(m-1)}, \tilde{d}) \rightarrow \overline{u}^{(m)}\) or \(t^{(m)}(\tilde{e}^{(m)})\), and consider how they must have been derived. Owing to the assumed absence of redex constructors in \(t\), the premises of these derivations must then be \(d_1 \rightarrow d'_1, d'_1 \rightarrow d''_1, \ldots, d_{1}^{(m-1)} \rightarrow \overline{u}^{(m)}\) (for some successor \(d'_1\) of \(d_1\), etc). They occur at branches of the effect-trees \(m_1, m_2, \ldots\) respectively of \(d_1\); hence, the successor terms \(t(d_1, \tilde{d}), \ldots, t(d_{1}^{(m-1)}, \tilde{d})\) (and \(\overline{u}^{(m)}\) or \(t^{(m)}(\tilde{e}^{(m)})\)) must also occur at these branches.

The resulting \(n\)-step behaviour of the term \(t(d_1, \tilde{d})\) may be represented in a by the following diagram, in analogy to the Diagram 3.1. (We are not asserting that \(t(d_1, \tilde{d})\) is a denotation; but it is convenient to represent its transition behaviour in the same notation.) We do not need to consider the behaviour of the leaves \(t^{(m)}(\tilde{e}^{m})\), so we omit effect-tree triangles corresponding to their behaviours.

```
\[
\begin{array}{c}
  t(d_1, \tilde{d}) \\
  \downarrow m_1 \\
  t(d'_1, \tilde{d}) \\
  \downarrow m_2 \\
  t'(\tilde{e}') \\
  \downarrow \overline{u}' \\
  t(d''_1, \tilde{d}) \\
  \downarrow m_3 \\
  t''(\tilde{e}'') \\
  \downarrow \overline{u}'' \\
  t(d'''_1, \tilde{d}) \\
  \downarrow \overline{u}''' \\
  \vdots
\end{array}
\]
```

This completes our description of the \(n\)-step behaviour of the term \(t(d_1, \tilde{d})\) in the case where there are no redex-constructors in the nested first-positions of \(t\). We now consider the simpler case where the term \(t(\tilde{d})\) has a redex constructor in nested-first position, possibly introducing effects. In this case, the resulting effect-tree \(\delta_1\) in the term’s behaviour depends only on the syntax-structure of the term \(t\), and not the behaviour of the arguments \(\tilde{d}\). The immediate behaviours at each leaf of the tree are then derived according to the top-right case shown earlier, giving the behaviour shown below; unlike before, there will be no need to consider
successive transitions of the terms $t'(\tilde{e}')$, and we omit them.

\[
\begin{array}{c}
t(\tilde{d}) \\
\downarrow \delta_1 \\
t'(\tilde{e}') \\
\end{array}
\]

**From Single to Multiple Transitions in the Lower-horizontal Path**

Now let us consider the lower horizontal path $TD \xrightarrow{T\bar{\gamma}_\zeta} T(T_n\bar{D}) \xrightarrow{\text{omd}} \cdots \xrightarrow{B^{n-1}\text{omd}} B^n T(T_n\bar{D})$; this map first coarsens the denotations $d$ in terms $t(\tilde{d})$ by applying $\beta_\gamma$, giving coarse-grained denotations $f$ and the same term $t(\tilde{f})$ with new arguments. Then it exhibits $n$ steps of effectful behaviour $t \rightarrow \delta(\tilde{b}) \rightarrow \delta((\epsilon_i(\tilde{b}_i))_{i \in I}) \rightarrow \text{etc.}$, in terms of the denotations $f$ at the leaves of $t$.

As above, we reason about the derivations of these transitions to represent them graphically; and we begin with the case where the term $t(d_1, \tilde{d})$ has no redex constructors in nested first position. We write $f_1$ for the result in $T_e\bar{D}$ of ‘coarsening’ $d_1$ by applying $\bar{\beta}_\zeta$, and similarly $g_1 = \bar{\beta}_\zeta(e_1)$, etc. The coarse-grained denotation $f_1$ may then be represented in terms of the diagram for $d_1$, as follows:

\[
\begin{array}{c}
f_1 \\
\downarrow m_1 \\
m_2 \\
\cdots \perp \\
(3, v''') \\
\end{array}
\]

Here, $m_i$ are again the effect-trees occurring in the behaviours of the $i^{th}$ successors of $d_1$; but now they have been joined into a single layer of effects. $(n, v)$ stands for an element of the initial algebra $\bar{D}$, terminating in $n$ steps with value $v$. The notation ‘$\cdots \perp$’ requires some explanation: recall that the effect-trees $m_i$ may be degenerate, containing no effects. In the denotation $d_1$, one may eventually encounter infinitely many such consecutive trees, whenever an execution-path introduces no effects, and also contains infinitely many transitions (such as the execution of `while (true) do {skip}`). One may check (by its least-fixpoint construction) that the map $\bar{\beta}_\gamma$ replaces those paths by $\perp$ directly. For convenience, we represent this information by simply allowing the trees $m_i$ to be degenerate, and use the given notation to indicate that $\perp$ may occur at any depth of the tree, whenever such an infinite execution path is encountered in $d_1$.

We may reason about the $n$-step behaviour of the term $t(f_1, \tilde{f})$ in a similar manner to before, by applying effectless derivation rules at each leaf of the effectful behaviour of $f_1$, retaining the conclusions in an effect-tree of the same shape. To do this, we relate the derivations to the
effect-free transitions of the original term: \( t(d_1, \tilde{d}) \rightarrow \ldots \rightarrow t(d_1^{(m-1)}, \tilde{d}) \rightarrow u^{(m)} \) or \( t^{(m)}(\tilde{\varepsilon}^{(m)}) \). As derived from the premises \( d_1 \rightarrow d'_1, d'_1 \rightarrow d''_1, \ldots, d''_1^{(m-1)} \rightarrow u^{(m)} \).

First note that for every leaf \( \varepsilon^{(m)} \) of the \( m \)th successors \( d_1^{(m)} \) of \( d_1 \), there is a corresponding leaf \( (m, \tilde{v}) \) of \( f_1 \). It has a matching series of effectless transitions: \( (m, \tilde{v}^{(m)}) \rightarrow (m - 1, \tilde{v}^{(m)}) \rightarrow \ldots \rightarrow \varepsilon^{(m)} \). This allows us to derive the following transitions:

\[
 t((m, \tilde{v}^{(m)}), \tilde{f}) \rightarrow \ldots \rightarrow t((1, \tilde{v}^{(m)}), \tilde{f}) \rightarrow \tilde{v}^{(m)} \text{ or } t^{(m)}(\tilde{g}^{(m)})
\]

This implies the behaviour of \( t(f_1, \tilde{f}) \) is as follows, where the \( \tilde{v}^{(m)} \) match the corresponding leaves of (3.1); \( t^{(m)} \) match the corresponding syntax-terms, and \( f_i^{(m)}, g_j^{(m)} \) are the image under \( T_{\beta} \) of \( d_i^{(m)}, e_j^{(m)} \). Note that after immediately introducing the effects within the denotation \( d_1 \), the leaves of this tree would undergo a series of effectless transitions before a terminal value \( \tilde{v}^{(m)} \) or new term \( t^{(m)}(\tilde{g}^{(m)}) \) is encountered (which may then introduce further effects); this information is represented explicitly on the diagram. As for the treatment of explicit divergence \( \downarrow \) in \( f_1 \), from a premise \( x \rightarrow \downarrow \), one would have an effect-free derivation of the transition \( t(x, \tilde{f}) \rightarrow \downarrow \) (according to the effect-free rules underlying the eEIC format); recall that such effect-free derivations may be carried out at each leaf of \( f_1 \) to deduce the behaviour of \( t(f_1, \tilde{f}) \).

\[
\begin{align*}
\ldots & \downarrow \\
& t((2, \tilde{v}''^{m}), \tilde{f}) \rightarrow t((1, \tilde{v}''^{m}), \tilde{f}) \rightarrow v''^{m} \\
& t((2, \tilde{v}''^{m}), \tilde{f}) \rightarrow t((1, \tilde{v}''^{m}), \tilde{f}) \rightarrow u''^{m} \\
& t((1, \tilde{v}''^{m}), \tilde{f}) \rightarrow v''^{m} \\
& t((1, \tilde{v}''^{m}), \tilde{f}) \rightarrow u''^{m} \\
& t'(\tilde{g}') \rightarrow \tilde{v}'^{m}
\end{align*}
\]

(3.2)

We now return to the simpler case where the term \( t(\tilde{d}) \) has a redex constructor in nested-first position. (Its behaviour is shown again, below-left.) The map \( T_{\beta} \) applies \( \tilde{\varepsilon} \) to the arguments \( \tilde{d} \), giving a term \( t(\tilde{f}) \). As the behaviour of this term does not depend on any arguments \( \tilde{f} \), we may apply exactly the same derivation as we did for \( t(\tilde{d}) \), but replacing \( \varepsilon \)'s with \( f \)'s (and \( e \)'s with \( g \)'s); this gives the behaviour shown below-right.

\[
\begin{align*}
& t(\tilde{d}) \quad \delta_1 \\
& \quad \quad \downarrow \delta_1 \\
& \quad \quad \quad \tilde{v}' \\
& t(\tilde{f}) \quad \delta_1 \\
& \quad \quad \downarrow \delta_1 \\
& \quad \quad \quad \tilde{v}'
\end{align*}
\]
Relating the Effect-trees Produced by Both Horizontal Paths

We now show the required induction step, assuming the induction hypothesis for all \( m < n \). We first consider the case where the \( t(d_1, \tilde{d}) \) contains no redex constructors in nested first position. The diagram (3.1) shows that after iterating the behaviour of \( t(d_1, \tilde{d}) \) for \( n \) steps, the resulting terminal values can come from two sources – either (1) from the terminal leaves \( \psi^{(m)} \) of \( d_1 \) (for \( m \leq n \)), resulting in terminal leaves \( \psi^{(m)} \) in the behaviour of \( t(d_1, \tilde{d}) \); or (2) from the unshown execution of successor terms, \( t^{(m)}(\tilde{e}^{(m)}) \).

Diagram 3.2 shows that for every terminal leaf \( \psi^{(m)} \) arising from the first case (1), there is a matching leaf in the behaviour of \( t(f_1, \tilde{f}) \) (given by applying \( \beta \zeta \)), and vice versa. As for the divergent leaves \( \cdots \perp \) in the behaviour of \( t(f_1, \tilde{f}) \), these correspond to sub-trees of \( d_1 \) which certainly do not introduce terminal values. Hence, the effect-trees exhibited by \( t(d_1, \tilde{d}) \) and \( t(f_1, \tilde{f}) \) agree up to terminal values.

For the second case (2), Diagram 3.2 also shows that the successor terms \( t^{(m)}(\tilde{e}^{(m)}) \) reachable in \( m \) steps by \( t(d_1, \tilde{d}) \) correspond (by applying \( \beta \zeta \) to arguments \( \tilde{e}^{(m)} \)) with terms \( t^{(m)}(\tilde{g}^{(m)}) \) reachable by \( t(\tilde{f}) \) in \( m \) steps. Any terminal leaves arising in \( n \) steps via these successors must arise in less than \( n \) steps when evaluating them directly; so the inductive hypothesis implies both horizontal paths agree at those leaves too.

Finally, if the behaviour of term \( t(\tilde{d}) \) depends on a redex constructor, then its behaviour differs from the behaviour of the term \( t(\tilde{f}) \) (given by the other horizontal path) simply by relabelling the arguments \( d \) into \( f \); the inductive hypothesis may then be applied to the successor terms \( t'(\tilde{e}'), t'(\tilde{g}') \) on the effect-trees produced by both paths. This tells us that for \( t(\tilde{d}) \) and \( t(\tilde{f}) \), the terminal leaves agree after \( n \) steps of behaviour along both horizontal paths.

Concluding This Chapter

This completes our syntactic proof of adequacy and compositionality for syntactic ETS’s, specified by effectful extensions under the eEIC format.

In this final section, we have reduced the problem of proving adequacy and compositionality for our Kleisli semantics to a large commuting diagram. Moreover, this reduction made no additional assumptions on the monad \( M \). We focused on the case of syntactic ETS’s, where the monad is given by syntactic effects \( T_e \), and gave a syntactic proof that the diagram commuted by exploiting the structure imposed by the eEIC rule format. In the following chapter, we will make use of the same reduction for general monads \( M \), but we will give a categorical description of several rule formats, and adapt the ideas of the previous theorem into a corresponding categorical proof of adequacy and compositionality.
Chapter 4

Semantics for Comodels and Effects

The semantic framework of the previous chapter, though largely categorical, was focused on the treatment of syntactic effects. In this chapter, we make the framework fully categorical, and apply it to languages whose semantics are described in terms of comodels and/or effects; we again illustrate with examples given by variants of the While language.

We begin by introducing several variants on the class of transition systems introduced in the previous chapter, the syntactic ets; they are given by $MB$-coalgebras for various monads $M$. We then review the technical constraints required to ensure existence of a final Kleisli coalgebra, in an analogous manner to the previous chapter; this is followed by an abstract formulation of the EIC rule format on the level of monads, which instantiates to concrete rule formats for the above transition systems – in contrast to the difficulty of interpreting the original abstract OS specifications of Section 3.1.2. Finally, we use the abstract EIC format to prove several results which guarantee adequacy and compositionality of our semantics for the different classes of transition systems. The main result is proved in the same way as in Section 3.4.4, through equality of two cones over the final sequence; however, the abstract EIC format permits a fully categorical proof.

4.1 Effectful and Comodel-based Languages As Coalgebras

4.1.1 The Semantic Effectful Transition System (ets)

We begin by addressing one major limitation of the theory in the previous chapter: the purely syntactic nature of the effects observed during program execution. We now assume we are given
a countable Lawvere theory $\mathcal{L}$ for the effects of the language (Definition 2.3.2), which exist identically at each sort. Recall that the category of models $\text{Mod}(\mathcal{L}, \mathcal{C})$ is given by countable product-preserving functors $\mathcal{L} \to \mathcal{C}$, with arrows given by natural transformations. If the underlying category $\mathcal{C}$ is locally countably presentable (l.c.p.), the forgetful functor $U^\mathcal{L} : \text{Mod}(\mathcal{L}, \mathcal{C}) \to \mathcal{C}$ has a left adjoint $F^\mathcal{L}$, which may be thought of as the free-model functor; and this induces a monad $U^\mathcal{L}F^\mathcal{L}$ on $\mathcal{C}$, which we will call $N_{e_0}$. As before, we exploit the fact that $\text{Cpo}_{\perp\uparrow}$ is l.c.p. because it is essentially algebraic (see [AR94] p.163).

The monad $N_{e_0}$ plays a similar role to the free $\Delta_\infty$-algebra monad $T_{e_0}$ of the previous chapter, which constructs syntactic effect-trees at a single sort. In concrete terms, $N_{e_0}$ may be thought of as constructing the equivalence classes $[\delta(\bar{x})]$ of effect-trees $\delta(\bar{x})$ over a single-sorted collection of variables $X$, quotiented by the equations described by the Lawvere theory. As before, in a multi-sorted setting the monad $N_{e_0}$ may be applied identically at each sort to give a monad $N_e$ on $\mathcal{C}^S$, with $(N_eX)_s = N_{e_0}(X_s)$. Note that a model of $\mathcal{L}$ in $\mathcal{C}^S$ is equivalent to an $S$-tuple of models $\text{Mod}(\mathcal{L}, \mathcal{C})$, and that the adjunction defining $N_{e_0}$ generalises componentwise to give an analogous adjunction for $N_e$.

This gives rise to a class of operational models for programming languages. Recall that a syntactic ETS assigns each program $p$ a syntactic effect-tree $\delta(\bar{b})$ whose leaves are given by terminal values $v$ or programs $p$; we may now consider a semantic ETS to instead assign the equivalence class $[\delta(\bar{b})]$ of such effect trees. Formally, the difference amounts to replacing $T_eB$-coalgebras with $N_eB$-coalgebras.

**Example 4.1.1.** As shown in [PP02], in categories with countable (co)powers, the free model of the theory of global store (with finitely many locations $L$), over variables $X$, has carrier $N_{e_0}X = (S \cdot X)^S$ where $S = \mathbb{N}_L$, $S \cdot X$ is an $S$-fold copower, and the superscript is an $S$-fold power; in particular, this holds in both $\text{Set}$ and $\text{Cpo}_{\perp\uparrow}$. In the context of $\text{Set}$, where $S$-fold copowers and products with $S$ coincide, $N_{e_0}$ is essentially the side-effect monad $M$; recall that we described an operational model for While in $\text{Set}$ as an $MB$-coalgebra $om : T0 \to (BT0 \times S)^S$ in Section 3.1.1. Hence, that operational model is an instance of a semantic ETS.

In the setting of $\text{Cpo}_{\perp\uparrow}$, one may interpret the free models $(S \cdot X)^S$ in a similar way, where $X$ is a cppo. The copower $S \cdot X$ is an $S$-fold coproduct, which may be described by pairs $(s, x)$ for $s \in S$ and $x \in X$, such that $(s, x) \subseteq (s', x')$ if and only if $s = s'$ and $x \subseteq x'$. Its $S$-fold power may be thought of as the collection of functions $f : S \to (S \cdot X)$ mapping initial stores to final stores and variables in $X$, with a pointwise ordering on the functions $f$; this makes explicit the analogy with the side-effect monad on $\text{Set}$.

**Example 4.1.2.** The carrier $N_{e_0}X$ of the free model of the theory of non-determinism in $\text{Set}$, over variables $X$, is the collection $\mathcal{P}_fX$ of finite subsets of $X$. By contrast, in the setting of $\text{Cpo}_{\perp\uparrow}$, given a cppo $X$, the free model $N_{e_0}X$ is instead the free convex powerdomain $\mathcal{P}_cX$ over $X$ ([AJ94] pp.93). In general, the structure of $\mathcal{P}_cX$ is rather involved. However, we will be
Chapter 4. Semantics for Comodels and Effects

primarily concerned with the free model $PcY$ over a flat cpso $Y$, which we may consider to arise by affixing a bottom element to some set $Z$; and this model is easier to describe (see [GS90] pp. 26). In the context of program execution, if we consider the set $Z$ to represent possible return values, then $PcY$ consists of countable collections of return values that may be exhibited by non-deterministic programs; and these collections are allowed to include a special value $\bot$, representing a divergent or ‘incomplete’ computation – so that a computation may non-deterministically diverge. For instance, a program which either returns 0 or 1, or diverges, would correspond to the collection of return values $\{0, 1, \bot\}$. 

Now we give a formal definition of $PcY$ on flat cpso $Y = Z\bot$. It consists of countable, non-empty subsets $S = \{z_i : i \in I\}$ of $Z \cup \{\bot\}$. The order structure of $PcY$ on subsets $S, S'$ is as follows. If $S$ does not contain $\bot$, then $S \sqsubseteq S'$ if and only if $S = S'$. However, if $S$ contains $\bot$, then $S \sqsubseteq S'$ if and only if $(S\setminus\{\bot\}) \subseteq S'$. (The bottom element of $PcY$ is the set $\{\bot\}$.)

Intuitively, the order structure of $PcY$ on these sets is as follows. If a subset $S$ of $Z \cup \{\bot\}$ contains the bottom element $\bot$, it is essentially ‘incomplete’; and one rises in the order on $PcY$ by extending ‘incomplete’ sets $S$ and/or ‘completing’ them by removing $\bot$. 

4.1.2 Converting a Syntactic ets into a Semantic ets

Although models of algebraic theories are well documented in $\mathbf{Set}$, it is less straightforward to characterise their models in ordered settings such as $\mathbf{Cpo}_\perp$ (see e.g. [Rob02] and [AJ94] Section 6); instead of attempting to give a concrete algebraic description of such models, we will rely on the structure of Lawvere theories to reason about them.

On this level, we will describe the process of ‘quotienting’ freely generated (and single-sorted) effect-syntax $Te_0X$ by the Lawvere theory, to give free models $Ne_0X$; this will be achieved through a map $\text{quot}^{(0)}_X : Te_0X \rightarrow Ne_0X$. Recall that, given a single-sorted signature $\text{Eff}$ of effect syntax, one obtains a corresponding syntax functor $\Delta_0X = \coprod_{e : s \rightarrow n \in \text{Eff}} X^n$, and that $Te_0X$ is the free $\Delta_0$-algebra functor; equivalently, $Te_0X$ is the initial $(X + \Delta_0)$-algebra. Hence, if we can also give $Ne_0X$ an $(X + \Delta_0)$-algebra structure, initiality implies existence of a map $Te_0X \rightarrow Ne_0X$ which we may take to be $\text{quot}^{(0)}_X$.

To achieve this, we begin by showing how any model $P$ of the Lawvere theory (in the base category, $\mathcal{C}$) induces a $\Delta_0$-algebra structure on its carrier $P1$ which interprets the effects. We identify the $n$-ary effect-syntax constructors in $\text{Eff}$ with arrows $e : n \rightarrow 1$ in the Lawvere theory. A model $P$ of the Lawvere theory maps $e$ to an arrow $P(e) : Pn \rightarrow P1$; we may pre-compose this arrow with an isomorphism $\theta_{Pn} : P1^n \cong Pn$ to be defined shortly, giving an arrow $P1^n \rightarrow P1$ which interprets the effect $e$ on the carrier $P1$. Finally, we take an $\text{Eff}$-indexed
coproduct $\gamma_P$ over these interpretations for all $e \in \text{Eff}$, giving a $\Delta_0$-algebra-structure

$$\gamma_P : \Delta_0P1 \to P1 \quad \gamma_P := [P(e) \circ \theta_{P,n}]_{e:a^n \to s} \in \text{Eff}. \quad (1)$$

Now we use the fact that $N_{e_0}X = U^eF^e X = (F^e X)(1)$ is the carrier of a free model $F^e X$ of the Lawvere theory; hence $N_{e_0}X = (F^e X)(1)$ has a $\Delta_0$-algebra structure $\gamma_{F^e X}$. Taking its coproduct with the monad unit $\eta_{X}^{N_{e_0}} : X \to N_{e_0}X$, we obtain a $(X + \Delta_0)$-algebra structure on $N_{e_0}X$, and hence the required arrow $\text{quot}_{X}^{(0)} : T_{e_0}X \to N_{e_0}X$ by initiality of $T_{e_0}X$.

It only remains to provide the isomorphism $\theta_n : P1^n \cong Pn$, which exists by the following argument. The object $n$ in the Lawvere theory is the $n$-fold product of 1, with projections $p_r : n \to 1$ for all $0 \leq r < n$ (as described in Section 2.3.2); hence, by product preservation $Pn$ is an $n$-fold product of $P1$, with projections $P(p_r) : Pn \to P1$. As they are both limits, there is an isomorphism $\theta_{P,n} : P1^n \cong Pn$ mediating between the preserved projections $P(p_r) : Pn \to P1$ and the standard product-projections $\pi_r : P1^n \to P1$; i.e. $P(p_r) \circ \theta_{P,n} = \pi_r$ for all $0 \leq r < n$.

**Example 4.1.3.** As described in Section 2.3.2, the free model of global store over an object $X$ has carrier $(S \cdot X)^S$. The interpretation of an update operation $\text{wr}_{x,n}$ on this model is as follows: let $f$ be a function of type $(S \cdot X)^S$. Then $\text{wr}_{x,n}(f)$ is the function of type $(S \cdot X)^S$ given by “precomposing $f$ with an update” as follows: $(\text{wr}_{x,n}(f))(s) = f(s[x \mapsto n])$. Given a $\mathbb{N}$-indexed collection of these functions $f$, the $\mathbb{N}$-ary interpretation of lookups $\text{rd}_x(f_1, f_2, \ldots)$ is the function which essentially takes a store $s$, looks up the value $s(x)$ of $x$, and returns $f_{s(x)}(s)$.

We may take the syntactic effect-signature $\text{Eff}$ to consist of variable lookups, corresponding to an $L$-indexed family of arrows $\text{rd}_x : \omega \to 1$, and updates, corresponding to a $(\mathbb{N} \times L)$-indexed family of arrows $\text{wr}_{x,n} : 1 \to 1$; then $\text{Eff}$ induces a syntax functor

$$\Delta X = L \cdot X^\omega + (\mathbb{N} \times L) \cdot X$$

and the interpretations of lookups and updates make the free model $(S \cdot X)^S$ into a $\Delta_0$-algebra.

The resulting $\Delta_0$-algebra morphism $\text{quot}_X : T_{e_0}X \to N_{e_0}X$ maps effect-trees over $X$ to their corresponding state-manipulation (and return value in $X$). For example, letting $u, v$ be variables in $L$, one has:

$$\text{quot}_X : \quad \text{rd}_u(\text{wr}_{e,0}(x_0), \text{wr}_{e,1}(x_1), \text{wr}_{e,2}(x_2), \ldots) \quad \mapsto \quad \lambda s. (x_{s(u)}, s[v \mapsto s(u)]).$$

**Example 4.1.4.** For the theory of non-determinism, the carrier of the free model in $\text{Cpo}_{+1}$ over a flat cppo $Y = Z_\bot$ is the convex powerdomain $P_Y$ of Example 4.1.2; the interpretation of the binary $\text{or}$ operator on non-empty subsets $S, S' \subseteq Z_\bot$ is set-union. The quotienting map $\text{quot}_X$ maps a (possibly infinite) syntactic $\text{or}$-tree, such as $\bot \text{ or } x_1 \text{ or } x_2 \text{ or } \ldots$, to the corresponding subset $S$ in $P_Y$ containing the distinct leaves of the tree, i.e. $\{\bot, x_1, x_2, \ldots\}$. (The process
of ‘completing’ a subset \( S \) – by removing \( \perp \) – may thus be interpreted in terms of syntactic or-trees, as replacing \( \perp \)-leaves with values \( x \) already occurring elsewhere in the effect-tree – e.g.

\[
\{ x, \perp \} = \text{quot}_X (x \text{ or } \perp) \subseteq \text{quot}_X (x \text{ or } x) = \{ x \}.
\]

The quotient \( \text{quot}^{(0)} \) is coherent with the \( \Delta_0 \)-algebra structures on both \( T_{e_0} \) and \( N_{e_0} \), in the sense that it is a monad morphism.

**Proposition 4.1.5.** The maps \( \text{quot}^{(0)}_X : T_{e_0}X \rightarrow N_{e_0}X \) define a monad morphism – meaning that the following diagrams commute:

\[
\begin{array}{ccc}
X & \xrightarrow{\eta_X} & T_{e_0}X \\
\downarrow & & \downarrow \\
N_{e_0}X & \xrightarrow{\text{quot}^{(0)}_X} & N_{e_0}X
\end{array}
\quad
\begin{array}{ccc}
T_{e_0}X & \xrightarrow{\eta_X} & T_{e_0}N_{e_0}X \\
\downarrow & & \downarrow \\
T_{e_0}X & \xrightarrow{\text{quot}^{(0)}_X} & N_{e_0}X
\end{array}
\]

**Proof.** (For convenience, we omit the superscript \( \mathcal{L} \) of the functors \( F^\mathcal{L}, U^\mathcal{L} \).)

It is convenient to make use of the characterisation of the initiality property of \( T_{e_0}X \) introduced at the end of Section 2.1.2: for any \( \Delta_0 \) algebra \( \nu : \Delta_0 Y \rightarrow Y \), there is a unique morphism \( ! \) making the left-hand diagram below commute. In particular, the quotienting map \( \text{quot}^{(0)}_X \) is the unique morphism making the right-hand diagram commute.

\[
\begin{array}{ccc}
\Delta_0 T_{e_0}X & \xrightarrow{\Delta_0 !} & \Delta_0 Y \\
\downarrow & & \downarrow \nu \\
T_{e_0}X & \xrightarrow{!} & Y \\
\downarrow & & \downarrow \\
X & \xrightarrow{\eta_X} & T_{e_0}X
\end{array}
\quad
\begin{array}{ccc}
\Delta_0 T_{e_0}X & \xrightarrow{\Delta_0 \text{quot}^{(0)}_X} & \Delta_0 N_{e_0}X \\
\downarrow & & \downarrow \tau_{FX} \\
T_{e_0}X & \xrightarrow{\text{quot}^{(0)}_X} & N_{e_0}X \\
\downarrow & & \downarrow \\
X & \xrightarrow{\eta_X} & T_{e_0}X
\end{array}
\]

Proving that the arrows \( \text{quot}^{(0)}_X : T_{e_0}X \rightarrow N_{e_0}X \) define a monad morphism requires checking commutativity of the two ‘monad-morphism’ diagrams for the unit and multiplication of \( T_{e_0}, N_{e_0} \), as well as naturality. The first diagram has already been shown to commute, as it the bottom triangle in the right-hand diagram above. We now focus on the diagram for multiplication, as naturality is similar and easier. We will prove that both paths satisfy the initiality property (as shown above-left) of the unique \( \Delta_0 \)-algebra morphism from \( T_{e_0}^2X \) into \( N_{e_0}X \) (with structure \( \theta_X \)), and so must coincide. This property is trivial to verify for the bottom path if we use the fact that \( \mu_X^{T_{e_0}} \) is a \( \Delta_0 \)-algebra morphism (as it is a \( T_{e_0} \)-algebra morphism, and these correspond with \( \Delta_0 \)-algebra morphisms: see [Tur96] pp.42-44). The property for the top path is represented below.
The left-most squares commute by naturality of $\psi : \Delta_0 T_{e_0} \Rightarrow T_{e_0}$ (from the adjunction generating $T_{e_0}$) and $\eta T_{e_0} : Id \Rightarrow T_{e_0}$. The middle parts commute by definition of $\text{quot}_{N_{e_0}X}^{(0)}$. It remains to prove commutativity of the right-most square. We recall that $\Delta_0 X = \coprod_{e : x^n \rightarrow x} X^n$ and that the arrows $\gamma_{FX}$ and $\gamma_{FN_{e_0}X}$ are defined on each component of this coproduct by $(FX)(e) \circ \theta_{FX,n}$ and $(FN_{e_0}X)(e) \circ \theta_{FN_{e_0}X,n}$ respectively. Hence, we may verify commutativity of the components separately, as shown below; taking the Eff-fold coproduct of these diagrams for all $e$ gives commutativity of the right-most square above as required. (We have flipped the square across the diagonal.)

\[
\begin{array}{cccc}
\Delta_0 T_{e_0} X & \Delta_0 T_{e_0} \text{quot}_{X}^{(0)} & \Delta_0 T_{e_0} N_{e_0} X & \Delta_0 N_{e_0} X \\
\psi T_{e_0} X & \psi N_{e_0} X & \gamma_{FN_{e_0}X} & \gamma_{FX} \\
T_{e_0} X & T_{e_0} \text{quot}_{X}^{(0)} & T_{e_0} N_{e_0} X & N_{e_0} X \\
T_{e_0} \eta_{X} & \text{quot}_{X}^{(0)} & \eta_{X} & \text{id} \\
\eta_{X} & \eta_{X} & \eta_{X} & \eta_{X} \\
X & N_{e_0} X & N_{e_0} X & N_{e_0} X
\end{array}
\]

The outermost equalities use the fact that the monad multiplication $\mu_{N_{e_0}} : N_{e_0} X \rightarrow N_{e_0} X$ may be defined to be $U \varepsilon_{FX} = (\varepsilon_{FX})_1$, where $\varepsilon : FU \Rightarrow Id$ is the counit of the adjunction $F \dashv U$; it is a natural transformation in the category of models $\text{Mod}(\mathcal{L}, \mathcal{C})$. Note that its component $\varepsilon_{FX}$ is a natural transformation between the models $FUFX$ and $FX$, which is why the third square commutes.

It remains to prove commutativity of the second square. We do this using the fact that the object $(FX)(n)$ has the universal property of an $n$-fold product of $(FX)(1)$, with projections $(FX)(p_j)$ for $j = 1, \ldots, n$ (as the model $FX$ is product-preserving). The object $((FUFX)(1))^n$, with projections $\pi_j : ((FUFX)(1))^n \rightarrow (FUFX)(1)$, allows us to define a cone over the $n$-fold product diagram of $(FX)(1)$, by the compositions $(\varepsilon_{FX})_1 \circ \pi_j : ((FUFX)(1))^n \rightarrow (FX)(1)$. Hence, there is a unique map $! : ((FUFX)(1))^n \rightarrow (FX)(n)$ mediating between this cone and the limit cone given by $(FX)(n)$ and $(FX)(p_j)$, i.e. satisfying $(FX)(p_j) \circ ! = (\varepsilon_{FX})_1 \circ \pi_j$ for $j = 1, \ldots, n$. 

\[
\begin{array}{cccc}
(N_{e_0} X)^n & (UFUFX)^n & (FUFX)(e) & (FUFX)(1) \\
\theta_{FUFX,n} & (\theta_{FUFX,n}) & (\varepsilon_{FX})_n & (\epsilon_{FX})_1 \\
((FX)(1))^n & (FX)(n) & (FX)(1) & UFX \rightarrow N_{e_0} X
\end{array}
\]
We will show that this property is satisfied by both paths in the second square above, and so they must coincide, completing the proof of commutativity of the second monad-morphism diagram. This will follow from commutativity of the following diagrams; from top-to-bottom and left-to-right, the parts commute for the following reasons: (1) definition of $\theta_{FUFX,n}$; (2) $\varepsilon_{FX}$ is a natural transformation between models $FUFX$ and $FX$; (3) by definition of the $n$-fold product $(f)^n$ of a morphism $f$; (4) definition of $\theta_{FX,n}$.

$$
\begin{array}{ccc}
(FUX)(1) & (FX)(1) & (FX)(1)\\
\downarrow \varepsilon_{FX,1} & \downarrow \varepsilon_{FX,1} & \downarrow \varepsilon_{FX,1}\\
(FUX)(p_j) & (FX)(p_j) & (FX)(p_j)\\
\downarrow \theta_{FUFX,n} & \downarrow \theta_{FUFX,n} & \downarrow \theta_{FUFX,n}\\
((FUFX)(1))^n & ((FX)(1)) & ((FX)(1))^n
\end{array}
$$

In a multi-sorted setting, we have assumed effects occur identically at each sort, so that the effect-monads $T_e$ and $N_e$ are given componentwise by $T_{e0}$ and $N_{e0}$ respectively. Hence, we define a multi-sorted quotient $quot : T_e \Rightarrow N_e$ by $(quot_X)_s = quot_X^0$, and it is easily shown to be a monad morphism between $T_e$ and $N_e$, because $quot_X^0$ is a monad morphism between $T_{e0}$ and $N_{e0}$.

Given a syntactic ETS $(X, \gamma : X \rightarrow T_eBX)$, the quotient $quot$ gives rise to a semantic ETS $(X, \gamma' : X \rightarrow N_eBX)$ by post-composing the $T_e$-coalgebra structure $\gamma$ with $quot_{BX} : T_eBX \rightarrow N_eBX$. We illustrate with an example, after introducing some useful notation conventions, in a multi-sorted setting, for (co)powers with a single-sorted object.

**Remark 4.1.6.** Given an object $X$ in the multi-sorted setting of $C^S$ and a set $A$, we will overload the notations $X^A$ and $A \cdot X$ to mean respectively the $A$-fold power and copower of $X$, which are given componentwise by the (co)powers of each component of $X$: e.g. $(X^A)_s = (X_s)^A$. Similarly, if $C$ is symmetric $\otimes$-monoidal closed, the category $C^S$ inherits a monoidal product $X \otimes Y$ and exponential $(X)^Y$ given componentwise: $(X \otimes Y)_s = X_s \otimes Y_s$ and $(X^Y)_s = (X_s)^Y$.

Similarly, given an object $X$ of $C^S$ and a single-sorted object $Z$ of $C$, we have componentwise constructions $(Z \otimes X)_s = Z \otimes X_s$ and $(X^Z)_s = (X_s)^Z$. Equivalently, these may be considered as instances of the product and exponential in $C^S$ of the objects $X$ and $W$, where the latter is given by $(W_s) = Z$ for all $s$. We will shortly make use of these constructions where $Z = C1$ is the carrier of a comodel in $C$.

**Example 4.1.7.** We may apply the quotient $quot_{BX}$ to the syntactic ETS given by the operational model $om : T0 \rightarrow T_eBT0$ of the language $sEWhile$ (in $Cpo_{11}^S$). The resulting semantic
ets \( T_0 \rightarrow (S \cdot BT_0)^S \) consists of three arrows, one for each sort \( s \), which assigns to each \( s \)-sorted program \( p \in T_0 \) a function \( (S \cdot (\text{Vals}_+ \oplus BT_0))^S \) describing the (one-step) stateful transition behaviour of that program. For instance, the program \( \text{wr}_{x,5}(\text{skip}) \) is assigned the effect-tree \( \text{wr}_{x,5}(\text{inr}(\text{skip})) \) in the syntactic ets given by \( \text{sWhile} \); whereas in the corresponding semantic ets, it is assigned the behaviour \( \lambda s. (s[x \mapsto 5], \text{inr}(\text{skip})) \).

**Example 4.1.8.** Consider an effectful extension of \( \text{SWhile} \) with non-determinism, via a syntactic binary or-operator. The resulting syntactic ets, an operational model \( T_0 \rightarrow T_0 \) where \( \Delta_0 X = X^2 \), gives transitions such as

\[
(0 \; \text{or} \; 1) \Rightarrow 1 \rightarrow (0 \; \Rightarrow \; 1) \; \text{or} \; (1 \; \Rightarrow \; 1) \rightarrow \text{false} \; \text{or} \; \text{true}.
\]

Recall that free models of the theory of non-determinism correspond to convex powerdomains \( P_c X \). Thus, when we post-compose the operational model with the quotient \( \text{quot}_{BT_0} \), we obtain a semantic ets \( T_0 \rightarrow P_c BT_0 \) assigning each program in \( T_0 \) an element of \( P_c BT_0 \) – a set of behaviours \( BT_0 \) given either terminal values \( \text{Vals}_+ \) or new terms \( T_0 \) – and possibly including divergence. The above transition sequence would become:

\[
(0 \; \text{or} \; 1) \Rightarrow 1 \rightarrow \{0 \; \Rightarrow \; 1, 1 \; \Rightarrow \; 1\} \rightarrow \{\text{false}, \text{true}\}.
\]

### 4.1.3 The Comodel-Based Transition System (cts)

In Section 3.1.1, we described a conventional operational model for \( \text{While} \) as a transition system, with state-space given by pairs \( \langle p, c \rangle \) of a program-state \( p \), and a store \( c \) of variables – an element of \( S = N^L \) – and transitions of the form \( \langle p, c \rangle \Rightarrow \langle p', c' \rangle \) or \( \langle v, c' \rangle \) (in the case of termination).

This was described by an \( MB \)-coalgebra \( \langle P, \gamma \rangle \), where \( MX = (S \cdot X)^S \) is the side-effect monad adapted to a multi-sorted setting (see Remark 4.1.6) and \( BX = V + X \). It may be seen as an instance of a more general kind of transition system, where the notion of persistent store \( S \) is replaced by a comodel.

We assume we are given a comodel \( C \) of \( L \) in a symmetric monoidal closed category \( C \) – i.e. a coproduct-preserving functor \( L^{op} \rightarrow C \) – which may be considered to have state-space \( C^1 \); this will play the role that the stores \( S \) played in the side-effect monad. We may represent the transition behaviour of multi-sorted states \( (P_s)_{s \in S} \) by a function \( \gamma \), with \( s \)-component \( \gamma_s : P_s \otimes C^1 \rightarrow (V + P)_s \otimes C^1 \). As before, we may use the closed structure of \( C \) to curry this function and produce an arrow \( \gamma'_s : P \rightarrow ((V + P)_s \otimes C^1)^{C^1} \). In analogy to the side-effect monad, we define a single-sorted monad \( N_{c_0} \) on \( C \) with \( N_{c_0} X = (X \otimes C^1)^{C^1} \), which arises via the adjunction \( (\_ \otimes C^1) \dashv (\_)^{C^1} \) defining the closed structure of \( C \). We then define a monad \( N_c \) on \( C^S \) given componentwise by \( N_{c_0} \); by overloading of notation, we may again write
it as $N_c X = (X \otimes C1)^C1$ (Remark 4.1.6). (Equivalently, one may define $N_c$ by exploiting the symmetric monoidal structure of $C^S$ directly.)

The curried arrows $(\gamma'_s)_{s \in S}$ are then equivalent to the structure of an $N_c B$-coalgebra. We call such a coalgebra a comodel-based transition system, or cts.

Comodels of the theory of global store have been studied in the context of $\text{Set}$ [PS04]. We may exploit such information to produce comodels in $\text{Cpo}_{\perp!}$: every comodel of a theory in $\text{Set}$ gives rise to a comodel in $\text{Cpo}_{\perp!}$, as follows. We use the following adjunction, where $(-)_{\perp}$ maps a set to a flat cppo, and $U$ maps a cppo to the set of its elements (including the bottom element $\perp$).

$$\begin{array}{ccc}
\text{Set} & \xrightarrow{(-)_{\perp}} & \text{Cpo}_{\perp!} \\
\downarrow U & & \downarrow U \\
\end{array}$$

Left-adjoints are colimit-preserving, so they are countable coproduct-preserving (c.c.p.). Thus, given a comodel in $\text{Set}$ – i.e. a c.c.p. functor $\mathcal{L}^{op} \to \text{Set}$ – we may post-compose it with $(-)_{\perp}$ to obtain a new c.c.p. functor $\mathcal{L}^{op} \to \text{Cpo}_{\perp!}$, which is a comodel of the theory in $\text{Cpo}_{\perp!}$.

Example 4.1.9. As described at the end of Section 2.3.3, there is a canonical comodel for the theory of global store with carrier $S = N^L$. The above construction then gives us a comodel $C$ in $\text{Cpo}_{\perp!}$, with carrier $C1 = S_{\perp}$ and analogous implementations of variable lookup and update.

In Example 4.1.1, we showed that the standard operational model for While can be considered as a semantic ets, or an $N_c B$-coalgebra, in the setting of $\text{Cpo}_{\perp!}$. In fact, it may also be considered as a cts: with respect to the canonical comodel $C$ in $\text{Cpo}_{\perp!}$ described above, where $C1 = S_{\perp}$, the monad $N_c X = (X \otimes S_{\perp})^S_{\perp}$ may be shown to be equivalent to the monad $N_c X = (S \cdot X)^S$, as follows. We have that $X \otimes S_{\perp} \cong S_{\perp} \otimes X \cong S \cdot X$, and that the strict function-space $Y_{S_{\perp}}$ is equivalent to an $S$-fold product $Y^S$. This implies that semantic ets’s are equivalent to cts’s with respect to the canonical comodel for global store in $\text{Cpo}_{\perp!}$.

4.1.4 Converting an ets Into a cts

This section does not play a critical role in our semantic framework, but we include it as it provides a connection between the semantics of languages in terms of effects, and comodels.

Given a comodel $C$ for an effect theory $\mathcal{L}$, one may convert a syntactic or semantic ets into a cts; the comodel allows us to traverse or ‘consume’ the effect-trees produced by a program, as it provides us with implementations of these effects.

Example 4.1.10. From Section 2.3.3, recall that a comodel for global store provides an implementation for each lookup effect $rd_x$, in that it has a transition function $\text{lku}_x : C1 \to N \cdot C1$; it
assigns to each comodel-state $c$ a pair $\langle n, c' \rangle$ describing the value of $n$ in the store, and the new comodel-state $c'$ resulting from the lookup (which will be the same as $c$, by the co-equations for global store; see Section 2.3.4). It also has a transition function $\text{upd}_{x,n} : C1 \to C1$ implementing variable updates, assigning to each comodel-state $c$ a new state $c'$ where the update has been performed.

Suppose an sEWhile program exhibits the following behaviour, which for simplicity we take to be of type $T_eP$ for a suitable object $P$.

$$ p \rightarrow \text{rd}_x(\text{wr}_y,0(p_0), \text{wr}_y,1(p_1), \ldots) $$

We show how this effect-tree is traversed by the canonical comodel with state-space $S_{\perp} = (\mathbb{N}L)_{\perp}$, initially in some state $s$. The comodel implements lookups $\text{lku} : S_{\perp} \to \mathbb{N} \cdot S_{\perp}$ via the function $\text{lku}_x(s) = (s(x), s)$, and updates $\text{upd}_{x,n} : S_{\perp} \to S_{\perp}$ via the function $\text{upd}_{x,n}(s) = s[x \mapsto n]$. (Being arrows in $\text{Cpo}_{\perp!}$, both functions must take $\perp$ to itself.)

First, the read effect $\text{rd}_x$ is encountered; we evaluate $\text{lku}_x(s)$ to obtain a value $n$ for the variable $x$, and a ‘new’ store $s$ (which is unaffected by the lookup). We then proceed to the $n$th branch $\text{wr}_y,n(p_n)$ of the read effect. To handle the write effect $\text{wr}_y,n$, we evaluate $\text{upd}_{y,n}(s)$; this gives us a new store $s[y \mapsto n]$, with which we replace $s$, and proceed to the leaf $p_n$, which concludes the traversal.

This process maps each initial comodel state $s$ to a new state $s'$, and it also returns a leaf $p_n$ of the effect-tree; this assignment $s \mapsto (b_n, s')$ may be described as a function of type $(P \otimes C1)C1 = NcP$. Thus we have mapped a syntactic effect-tree in $T_eP$ into a comodel-manipulation in $NcP$. (Note that if the effect-tree in $T_eP$ was infinite-depth, as may happen in $\text{Cpo}_{\perp!}$, then the traversal may diverge for some comodel states; the previous function would then assign $s \mapsto \perp$.)

We now show that this traversal of effects by a comodel may be formalised by a monad morphism $m_P : T_eP \to NcP$. Given such a monad morphism $m$ and a syntactic ets -- a $T_eB$-coalgebra $(X, \gamma : X \to T_eBX)$ -- we may post-compose its coalgebra-structure with $m_{BX} : T_eBX \to NcBX$ to obtain an $NcB$-coalgebra, or cts. We define the monad morphism $m$ via the following lemma:

**Lemma 4.1.11.** Let $M, N$ be monads on a category $\mathcal{C}$, such that there are functors $U : D \to \mathcal{C}$ and $F, G : \mathcal{C} \to D$ with $F \dashv U$, $M = UF$, $N = UG$, and a natural transformation $n : GUG \Rightarrow G$ with $Un = \mu^N$. Then there is a monad morphism $m : M \Rightarrow N$.

**Proof.** We will obtain the monad morphism $m$ by first defining a natural transformation $p : F \Rightarrow G$; we will then show that $m = Up$ is a monad morphism. We may define each component $p_X : FX \to GX$ as the transpose of the unit of $N = UG$ across the adjunction $F \dashv U$: 

...
\[ \eta^N_X : X \to UGX \]
\[ (\eta^N_X)^2 : FX \to GX \]

Recalling the transpose of an arrow \( g : X \to UY \) can be expressed as \( \varepsilon_Y \circ Fg \), where \( \varepsilon \) is the counit of the adjunction, we may define \( p : F \Rightarrow G \) as the natural transformation \( \varepsilon_G \circ F\eta^N \).

We prove \( m = Up \) is compatible with the units of \( N \) and \( M \) – i.e. \( \eta^N = m \circ \eta^M \) – as follows:

\[
m \circ \eta^M = Up \circ \eta^M = U\varepsilon_G \circ UF\eta^N \circ \eta^M = U\varepsilon_G \circ \eta^M_{UG} \circ \eta^N = \eta^N
\]

The first two steps are by definition of \( m \) and \( p \), step 3 is naturality of \( \eta^M \) using the fact that \( M = UF \) (note that \( M \) and \( N \) have swapped!), and step 4 uses the fact that \( U\varepsilon \circ \eta^M_{GU} = \text{id} \) as \( F \dashv U \) is an adjunction.

It remains to prove that \( m \) is compatible with the multiplications of \( N \) and \( M \) – i.e. the left diagram below commutes. In fact it is the image under \( U \) of the right-hand diagram: recall that \( M \) is induced by the adjunction \( F \dashv U \), so its multiplication may be written in terms of the counit, viz. \( U\varepsilon_F = \mu^M \). Also, by assumption \( n \) is a natural transformation \( GUG \Rightarrow G \) such that \( Un = \mu^N \).

\[
\begin{array}{ccc}
M^2X & \xrightarrow{M\eta_X} & MNX \\
\downarrow & & \downarrow \\
MX & \xrightarrow{m_X} & NX
\end{array}
\quad
\begin{array}{ccc}
FUX & \xrightarrow{FU\eta_X} & FUGX \\
\downarrow & & \downarrow \\
FX & \xrightarrow{p_X} & GX
\end{array}
\quad
\begin{array}{ccc}
N^2X & \xrightarrow{N\eta_X} & NX \\
\downarrow & & \downarrow \\
FX & \xrightarrow{p_X} & GX
\end{array}
\]

In the right diagram above, the left square commutes by naturality of the counit \( \varepsilon \). The triangle may be broken down as follows. The top triangle is the definition of \( p_{UGX} \); the rightmost area is naturality of the counit \( \varepsilon : FU \Rightarrow \text{Id} \); and the small triangle is (\( F \) applied to) one of the monad laws for \( N \), viz. \( \mu^N_X \circ \eta^N_{NX} = \text{id}_{NX} \).

\[
\begin{array}{ccc}
FUGX & \xrightarrow{\varepsilon_{GUGX}} & GUGX \\
\downarrow & & \downarrow \\
FUG & \xrightarrow{F\mu^N_{GX}} & GUGX
\end{array}
\quad
\begin{array}{ccc}
FUG & \xrightarrow{\varepsilon_{GUGX}} & GUGX \\
\downarrow & & \downarrow \\
FUG & \xrightarrow{F\mu^N_{GX}} & GUGX
\end{array}
\quad
\begin{array}{ccc}
GUX & \xrightarrow{\varepsilon_{GX}} & GX \\
\downarrow & & \downarrow \\
GUX & \xrightarrow{\varepsilon_{GX}} & GX
\end{array}
\]

Using this lemma, the following proposition will show that there is a monad morphism \( m^F : N_e \Rightarrow N_c \) allowing us to convert a semantic ETS into a CTS. We instantiate the proposition with \( \mathcal{D} = \mathcal{C}^S \), exploiting the fact that in \( \mathcal{C}^S \), adjunctions (and hence symmetric monoidal-closure) are given componentwise by their counterparts in \( \mathcal{C} \); and that one may take an \( s \)-fold functor
product of a comodel in \( C : \mathcal{L}^{op} \to \mathcal{C} \), to obtain a comodel \( C' = \langle C, \ldots, C \rangle : \mathcal{L}^{op} \to \mathcal{C}^S \) in \( \mathcal{C}^S \). This makes the corresponding multi-sorted monad \( N_c X = (X \otimes C')^{C1} \) on \( \mathcal{C}^S \) - where the monoidal product \( \otimes \) and exponential are given componentwise - equivalent to the monad \( N_c X = (X \otimes C1)^{C1} \) (note the \( C \) rather than \( C' \)) defining the behaviour of \( \mathcal{C} \)'s, and using the overloaded notation of Remark 4.1.6 (for products and exponentials with the single-sorted object \( C1 \) in a multi-sorted setting).

This allows us to precompose the resulting monad morphism \( m^c \) with the quotienting map \( \text{quot} : T_c \Rightarrow N_c \) from Proposition 4.1.5, giving us a monad morphism \( T_c \Rightarrow N_c \). This allows us to convert the coalgebra structure of a syntactic \( \mathcal{C} \)'s into the structure of a \( \mathcal{C} \)'s.

**Proposition 4.1.12.** Let \( M = N_c \) be the monad on a symmetric monoidal closed category \( \mathcal{D} \) induced by an adjunction between \( \mathcal{D} \) and \( \text{Mod}(\mathcal{L}, \mathcal{D}) \), given by the left adjoint \( F \) to the forgetful functor \( U : \text{Mod}(\mathcal{L}, \mathcal{D}) \to \mathcal{D} \) (as before, we omit the superscripts \( .c \)). Also let \( N \times N = N_c X = (X \otimes C1)^{C1} \) be the monad induced by a comodel \( C \) of theory \( \mathcal{L} \) in \( \mathcal{D} \). Then Lemma 4.1.11 holds, so that there is a monad morphism \( m^c : N_c \Rightarrow N_c \).

**Proof.** We need to produce a functor \( G : \mathcal{D} \to \text{Mod}(\mathcal{L}, \mathcal{D}), \) mapping objects of \( \mathcal{D} \) to models of \( \mathcal{L} \) in \( \mathcal{D} \), such that \( UG = N_c \); and we need to provide a natural transformation \( n : GUG \Rightarrow G \) such that \( Un = \mu^N_c \). Note that \( n \) lives in the category of models; an arrow between models is simply a natural transformation between the corresponding functors \( \mathcal{L} \rightarrow \mathcal{D} \). Following the definition of countable Lawvere theories in terms of a skeleton \( N_1 \) of \( \text{Set} \) (Definition 2.3.1), we consider the objects \( n, m \) of \( \mathcal{L} \) to be countable sets; we write \( n + m \) for their disjoint union, which is the product of \( n \) and \( m \) in \( \mathcal{L} \), and the coproduct in \( \mathcal{L}^{op} \).

We now give a concrete definition of \( G \). The model \( GX \) maps a countable set \( n \) in \( \mathcal{L} \) to \( (X \otimes C1)^{Cn} \), and an arrow \( e : n \rightarrow m \) to \( (GX)(e) = (X \otimes C1)^{C(e^{op})} : (X \otimes C1)^{Cn} \rightarrow (X \otimes C1)^{Cm} \). We shortly show that \( GX \) is indeed a countable product-preserving (c.p.p.) functor, as required for it to be a model. Lastly, \( G \) itself takes arrows \( f : X \rightarrow Y \) in \( \mathcal{D} \) to natural transformations \( Gf : GX \Rightarrow GY \) between the model functors, where \( (Gf)_n = (f \otimes C1)^{Cn} : (X \otimes C1)^{Cn} \rightarrow (Y \otimes C1)^{Cn} \); it is straightforward to show that this makes \( G \) itself into a functor and we omit the details.

With this definition, we verify that \( UGX = (GX)(1) = (X \otimes C1)^{C1} = N_c X \). Hence, to provide the required natural transformation \( n : GUG \Rightarrow G \), note that \( GUGX = G(UGX) = GN_c X \) is a model mapping a countable set \( n \) to \( ((X \otimes C1)^{C1} \otimes C1)^{Cn} \); the arrow \( n_X \) (in the category of models) needs to be a natural transformation from the model \( GUX \) to the model \( GX \) which maps \( n \) to \( (X \otimes C1)^{Cn} \) - and this means a family of arrows \( (n_X)_m : ((X \otimes C1)^{C1} \otimes C1)^{Cm} \rightarrow (X \otimes C1)^{Cm} \).

The natural way to do this is to use the ‘evaluation’ evaly : \( Y^{C1} \otimes C1 \rightarrow Y \) given by the monoidal closed structure of the category \( \mathcal{D} \), and define \( (n_X)_m = (\text{eval}_{X \otimes C1})^{Cm} \); one may
verify that this definition is indeed natural in both \( X \) and \( m \). Finally, the forgetful functor \( U \) maps a natural transformation \( n_X \) (between models) to its \( 1 \)-component \( (n_X)_1 \); hence we have

\[
U n_X = (n_X)_1 \circ ((\text{eval}_{X_C^1})^C) : ((X \otimes C1)^C1 \otimes C1 \otimes C1) \to (X \otimes C1)^C1 = \mu_X^N
\]

so that all the requirements on the natural transformation \( n \) are met.

Lastly, to verify the requirement that \( GX \) is indeed c.p.p. and hence a model, we give it a functorial definition, making use of the adjunctions shown below (which come from the fact that \( D \) is symmetric monoidal-closed). We will exploit the fact that right adjoints preserve limits, thus \( (A)^- \) is countable product-preserving.

\[
\begin{align*}
&\begin{array}{cc}
D & (-) \otimes A \\
\perp & \rightarrow \\
(-)^A & D
\end{array} & \begin{array}{cc}
D & ((A)^-)^{op} \\
\perp & \rightarrow \\
(A)^- & D^{op}
\end{array}
\end{align*}
\]

To define \( GX \) functorially, we note that the comodel \( C : \mathcal{L}^{op} \to D \) is a countable coproduct-preserving functor; thus \( C^{op} : \mathcal{L} \to D^{op} \) is a countable product-preserving (c.p.p.) functor. Now postcomposing this with the c.p.p. functor \( (X \otimes C1)^- : D^{op} \to D \) gives a c.p.p. functor \( \mathcal{L} \to D \) which may be seen to be equivalent to \( GX \).

\[\square\]

### 4.1.5 The Comodel-and-Effect Based Transition System (cets)

Our final class of transition system combines features of both the previous classes. It is relevant to situations where programs are paired with comodels of a Lawvere theory \( \mathcal{L}_1 \) (like a cts), but execution may also exhibit effects from a different theory \( \mathcal{L}_2 \). We assume the two classes of effects are combined via the tensor \( \mathcal{L}_1 \otimes \mathcal{L}_2 \) of the theories. Our leading example is the tensor of the theory \( \mathcal{L}_1 \) of global store (which has a comodel) with the theory \( \mathcal{L}_2 \) of non-determinism (which does not, as discussed in Section 2.3.3).

In this situation, an operational model consists of configurations \( \langle p, c \rangle \) of a state \( p \) and a comodel-state \( c \), as in a cts. However, a transition may also introduce syntactic effects (or their equivalence classes), considered as branches in the computation; and each branch consists of a new configuration. We assume these effects come from a Lawvere theory \( \mathcal{L}_2 \), and that the comodel is for another theory \( \mathcal{L}_1 \).

For syntactic effects, the general form of a transition is then \( \langle p, c \rangle \to \delta((b_i, c_i)_{i \in I}) \) where \( \delta \) is an effect-tree, and the atomic behaviours \( b_i \) are either terminal values \( v_{i1} \), or successor states \( p_i \).
Informally, the semantic version replaces the effect-tree $\delta(\cdots)$ with its equivalence class $[\delta(\cdots)]$ under the Lawvere theory. We refer to such a transition system as a syntactic or semantic CETS respectively.

We introduce a non-deterministic variant of While as an example of this kind of transition system:

**Definition 4.1.13.** The language $\textsc{NDWhile}$ has the syntax of While, extended with a binary syntactic operator $\texttt{or}$ at each type.

**Example 4.1.14.** Here is a transition sequence illustrating the intended operational semantics of $\textsc{NDWhile}$, viewed as a syntactic CETS:

$$\langle x=(0 \texttt{ or } 1), s \rangle \rightarrow \langle x=0, s \rangle \texttt{ or } \langle x=1, s \rangle \rightarrow \langle \sharp, s[x \mapsto 0] \rangle \texttt{ or } \langle \sharp, s[x \mapsto 1] \rangle$$

We introduce the decoration $\cdot(2)$ to indicate that we are concerned with effects from the theory $L_2$; thus, we assume we are given a single-sorted signature $\text{Eff}$ of effects drawn from the theory $L_2$, with corresponding effect-syntax functor $\Delta^{(2)}_0$ and effect-syntax monad $T^{(2)}_e$. Applying this monad componentwise gives a syntactic effect-monad $T^{(2)}_e$ on $\mathcal{C}^1$. Similarly, we write $N^{(2)}_e$ for the monad given componentwise by the free models of $L_2$.

Analogously to cts’s and ets’s, we may now represent the transition behaviour of a CETS by an arrow $\gamma : P \otimes C^1 \rightarrow M((V + P) \otimes C^1)$, where we take the monad $M$ to be $T^{(2)}_e$ for a syntactic CETS, and $N^{(2)}_e$ for a semantic CETS. As before, assuming the category is symmetric $\otimes$-monoidal closed, we curry this arrow to obtain $\gamma' : P \rightarrow (M(BP \otimes C^1))^C^1$. This may be considered the structure of a coalgebra: we define the monads

$$T^{(2)}_e X = (T^{(2)}_e(X \otimes C^1))^C^1 \quad \text{and} \quad N^{(2)}_e X = (N^{(2)}_e(X \otimes C^1))^C^1$$

so that syntactic and semantic CETS’s are equivalent to $T^{(2)}_eB$- and $N^{(2)}_eB$-coalgebras respectively.

In the same way as we did for ets’s, we may convert a syntactic CETS into a semantic CETS using the monad morphism $\text{quot}_X : T^{(2)}_e X \rightarrow N^{(2)}_e X$ as defined in Section 4.1.2, with respect to the theory $L_2$; this quotients the syntactic effects from theory $L_2$ into semantic equivalence classes. This is achieved by post-composing the syntactic CETS structure $X \rightarrow T^{(2)}_eBX = (T^{(2)}_e(BX \otimes C^1))^C^1$ with the following map, which yields the structure $X \rightarrow N^{(2)}_eBX$ of a semantics CETS.

$$(\text{quot}_{(BX \otimes C^1)})^C^1 : T^{(2)}_e = (T^{(2)}_e(BX \otimes C^1))^C^1 \rightarrow (N^{(2)}_e(BX \otimes C^1))^C^1 = N^{(2)}_e.$$

It is straightforward to show that $(\text{quot}_{(\_ \otimes C^1)})^C^1$ is also a monad morphism $N^{(2)}_e \Rightarrow T^{(2)}_e$, using the fact that quot is a monad morphism.
Example 4.1.15. Consider the same transition sequence shown above for \texttt{NDWhile}. In the corresponding semantic CETS, the quotienting map \texttt{quot} replaces the syntactic or-trees with sets of behaviours, as shown:

\[
\langle x=(0 \text{ or } 1), s \rangle \rightarrow \{ \langle x=0, s \rangle, \langle x=1, s \rangle \} \rightarrow \{ \langle *, s[x \mapsto 0] \rangle, \langle *, s[x \mapsto 1] \rangle \}
\]

4.1.6 Converting an ETS into a semantic CETS

In analogy to the conversion of CTS’s to ETS’s, given a syntactic effect-tree \( \delta(\tilde{x}) \) drawn from the effects of the combined theory \( L_1 \otimes L_2 \), one may use a comodel of theory \( L_2 \) (in some initial state \( c \)) to consume some of the effects \( \delta \). Whenever an effect \( e(\cdots) \) from the other theory \( L_1 \) is encountered, one propagates the comodel-state to each branch of \( e \) and continues, eventually producing an effect-tree \( e(\langle x_i, c_i \rangle_{i \in I}) \) of pairs, containing leaves \( x_i \) of the original tree and new comodel-states \( c_i \).

Example 4.1.16. We consider effects given by the tensor of global store with non-determinism, and the comodel for global store given by \( S = N^L \). Then a traversal of the effect-tree

\[
\text{wr}_{x,5}(p_1) \text{ or } (p_2 \text{ or } \text{wr}_{x,6}(p_3))
\]

starting with comodel-state \( c \), would yield the tree

\[
\langle p_1, c[x \mapsto 5] \rangle \text{ or } (\langle p_2, c \rangle \text{ or } \langle p_3, c[x \mapsto 6] \rangle).
\]

We write \( T_e \) for the syntactic monad given by a set of effects from \( L_1 \otimes L_2 \), and similarly \( N_e \) for the monad given by free models of the tensor theory. Now we may describe the traversal process formally by an arrow \( T_e P \rightarrow N_{ce} P = (N_e(2)(P \otimes C1))^{C1} \), from syntactic effect-trees into (equivalence classes of) effect-trees over comodel-states, parameterised by the initial comodel state. As before, we achieve this by defining a monad morphism \( m_{ce} : N_e \Rightarrow N_{ce} \) through the following proposition (taking \( D = C^{S} \) again, and the \( S \)-fold product of a comodel in \( C \), as we did in Proposition 4.1.12). By precomposing with the monad morphism \( \text{quot} : T_e \Rightarrow N_e \) given by Proposition 4.1.15, these maps permit conversion of both syntactic and semantic ETS’s into semantic CETS’s.

Proposition 4.1.17. Let \( D \) be a symmetric monoidal closed, and locally countably presentable, category, and let \( L = L_1 \otimes L_2 \) be a tensor of two countable Lawvere theories as shown. Let \( M = N_e \) be the monad induced by the left adjoint \( F \) to the forgetful functor \( U : \text{Mod}(L, \otimes L_2, D) \rightarrow D \), and let \( NX = N_{ce}X = (N_e(2)(X \otimes C1))^{C1} \) be the monad induced by a comodel \( C \) of theory \( L_1 \). Then there is a monad morphism \( m_{ce} : N_e \Rightarrow N_{ce} \).
Proof. We exploit the coherent equivalence \((P : A \to B, \ Q : B \to A)\) between the category of models of the combined theory \(A = \text{Mod}(\mathcal{L}_1 \otimes \mathcal{L}_2, \mathcal{D})\) and the category of ‘models of models’ \(B = \text{Mod}(\mathcal{L}_2, \text{Mod}(\mathcal{L}_1, \mathcal{D}))\) of Theorem 2.3.4. This is equivalent(!) to an adjoint equivalence \(P \dashv Q\), an adjunction where the unit and counit \(\eta, \varepsilon\) are natural isomorphisms. Coherence means that the functors \(P, Q\) preserve the carriers of the models [HPP06]. This amounts to asserting that \(U = U_2 \circ U_{12} \circ Q\) in the following diagram, where \(F_2\) is the left adjoint to the forgetful functor \(U_2\) (so that \(N^e(2) = U_2 F_2\)) and \(F_{12}\) is the left adjoint to the forgetful functor \(U_{12}\) on the category of models, as shown.

![Diagram](attachment:image.png)

The left adjoint \(F_2\) exists because \(\mathcal{D}\) is locally countably presentable (l.c.p.); but the category of models \(\text{Mod}(\mathcal{L}_2, \mathcal{D})\) is also l.c.p., so by the same argument \(F_{12}\) exists [HPP06]. As left adjoints are unique up to isomorphism, the left adjoint \(F\) of \(U\) is isomorphic to \(PF_{12} F_2\); so up to isomorphism, we may take \(M = N^e = UF = U_2 U_{12} Q \circ PF_{12} F_2\).

By removing the equivalence functors \(QP\) from the monad \(M\), let monad \(M'\) be defined as \(U_2 U_{12} F_2 F_2\). We will use Lemma 4.1.11 to show that there is a monad morphism \(m' : M' \Rightarrow N^e\). Then, noting that \(QP\) is a monad, we use the trivial fact that the inverse \(\eta^{-1} : QP \Rightarrow \text{Id}\) of the unit of the adjunction is a monad morphism from \(QP\) to the identity functor. By ‘inserting it into the adjunction’, we obtain the following natural transformation, which is easily shown to be a monad morphism \(e : M \Rightarrow M'\).

\[
e : U_2 U_{12} \eta_{F_{12} F_2}^{-1} : \quad M = U_2 U_{12} Q P F_{12} F_2 \quad \Rightarrow \quad U_2 U_{12} F_2 F_2 = M'.
\]

We may post-compose it with the monad morphism \(m' : M' \Rightarrow N^e\) to be obtained shortly, giving us the sought-after monad morphism \(m = m' \circ e : M \Rightarrow N^e\).

For convenience, let us define \(U' = U_2 \circ U_{12}\). To obtain \(m'\), we aim to define a functor \(G : \mathcal{D} \to \text{Mod}(\mathcal{L}_1, \text{Mod}(\mathcal{L}_2, \mathcal{D}))\) such that \(U'GX = N^e X = (N^e(2)(X \otimes C1))^C1\), and a natural transformation \(n : GU'G \Rightarrow G\) such that \(U'n = \mu^{N^e}\). As in Proposition 4.1.12, we do this concretely: we define the model \(GX\) on objects \(n_1\) of \(\mathcal{L}_1\) as follows. Note that \((GX)(n_1)\) is itself a model in \(\text{Mod}(\mathcal{L}_2, \mathcal{D})\), defined on objects \(n_2\) of \(\mathcal{L}_2\).

\[
GX : n_1 \mapsto ((GX)(n_1) : n_2 \mapsto ((F_2(X \otimes C1))(n_2))^{C n_1})
\]

(We omit details of the action of \(G\) on arrows \(f : X \to Y\); the components \((Gf)_{n_2}\) of \(Gf\) are themselves natural transformations between the models \((GX)(n_2)\) and \((GY)(n_2)\), with components \(((Gf)_{n_2})_{n_1}.)
We will shortly show that both $GX$ and its components $(GX)(n_1)$ are indeed countable product-preserving (c.p.p.), as required for models. First, we show that the above definition satisfies the first requirement of Lemma 4.1.11: we have

$$U'GX = ((GX)(1))(1) = ((F_2(X \otimes C1))(1))^C1 = (U_2F_2(X \otimes C1))^C1 = (N_e^{(2)}(X \otimes C1))^C1.$$ 

Now we define the components of the natural transformation $n_X : GU'GX \to GX$. Note that by the previous fact, we have

$$GU'GX : n_1 \mapsto \left( n_2 \mapsto \left( (F_2((N_e^{(2)}(X \otimes C1))^C1 \otimes C1))(n_2) \right)^{Cn_1} \right)$$ 

and so we define the components of $n_X$ as follows, where we write eval for the evaluation map arising from the closed structure of $D$, and $\varepsilon$ for the counit of the adjunction $F_2 \dashv U_2$.

$$((n_X)_1)_{n_2} : ((\varepsilon_{F_2(X \otimes C1)}(n_2))^{Cn_1} \mapsto \left( (F_2((N_e^{(2)}(X \otimes C1))^C1 \otimes C1))(n_2) \right)^{Cn_1} = (F_2U_2F_2(X \otimes C1))(n_2))^{Cn_1} \mapsto ((F_2(X \otimes C1))(n_2))^{Cn_1}$$ 

This definition allows us to check that $U'n_X = \mu_X$ as required. We have $U n_X = U_2U_1 n_X = ((n_X)_1)_{1}$; we use the fact that for objects $Y$ and arrows $Y \to Z$, we have $(F_2Y)(1) = U_2F_2Y = N_e^{(2)}Y$ and similarly $(F_2f)(1) = N_e^{(2)}f$. In applying this, we also have that $U_2\varepsilon F_2Y$ is the multiplication $\mu_Y^{(2)}$ of $N_e^{(2)}$; thus the above composition may be written as follows, which is the multiplication of $N_{ce}$ as required.

$$((n_X)_1)_{:} : \left( (N_e^{(2)}((N_e^{(2)}(X \otimes C1))^C1 \otimes C1))^C1 \mapsto \left( (N_e^{(2)}(X \otimes C1))^C1 \mapsto \left( \mu_{(X \otimes C1)} \right) \mapsto \left( N_e^{(2)}(X \otimes C1))^{C1} \right) \right)$$ 

We now show that $GX$ and $(GX)(n_1)$ are indeed c.p.p. For each object $n_1$, the functor $(GX)(n_1)$ may be defined as the composition $(-)^{Cn_1} \circ F_2(X \otimes C1)$, and both functors are c.p.p. (as the latter is a model, and the former is a right adjoint – see Proposition 4.1.12).

To show that $GX$ is also c.p.p., we need to show that for any countably-indexed collections of objects $(n_i)_{i \in I}$ of $L_1$, $(GX)(\prod_{i \in I} n_i)$ is the product of the models $(GX)(n_i)$ for all $i \in I$. To characterise this product, we use the routinely provable fact that products in a category of models are given componentwise: i.e. given a countable $J$-indexed collection of models $K_j$ of $\text{Mod}(L_2, D)$, we may define their product $\prod_{j \in J} K_j$ on objects $n_2$ of $L_2$ by $(\prod_{j \in J} K_j)(n_2) =$
\[ \prod_{j \in J} (K_j(n_2)), \text{ and similarly on arrows of } \mathcal{L}_2. \] Hence, to show that \( GX \) is c.p.p. (on \( \mathcal{L}_1 \)), we may show that for all \( n_2 \) and countable collections \( (n_i)_{i \in I} \), \( (GX)(\prod_{i \in I} n_i)(n_2) = \prod_{i \in I} (GX)(n_i)(n_2) \). To do this, we may prove that the following functor is c.p.p. for each \( n_2 \):

\[ (GX)(-) (n_2) : \mathcal{L}_1 \to \mathcal{D} \quad n_1 \mapsto ((F_2(X \otimes C1))(n_2))^Cn_1 \]

This follows from the same reasoning as Proposition 4.1.12, namely that it is a composition \((F_2(X \otimes C1))^C \circ C^{op}\) of two c.p.p. functors, a right-adjoint and the \( op \) of a comodel.

4.2 Evaluation-In-Context Congruence (EIC) formats for Effectful and Stateful Languages

In Section 3.4.1, we introduced a syntactic restriction on effectless operational rules – the eEIC format of Definition 3.4.4 – so that when the specification was extended to incorporate effects, our final-coalgebra semantics would be adequate and compositional. Example 3.4.3, demonstrating why such a restriction was needed, is readily adapted to the transition systems we have introduced above; and so they too require a restriction on operational specifications.

In principle, one may try to adapt the idea of ‘effectful extension’, and the proof of Theorem 3.4.7, to languages with comodels and/or effects; but it would be better to seek a unified semantic approach for all the different classes of transition system, just as we have framed all their operational models as \( MB \)-coalgebras by varying the monad \( M \). In this section, we adapt the idea of evaluation-in-context to provide congruence formats for the other transition systems cts and cets; we will give syntactic presentations of the formats before their categorical definitions, and show that they may be considered as instances of a single, monadic definition. This involves imposing a suitable structure on the syntax functor \( \Sigma \) of the language.

Context and Redex Terms

The key idea of the eEIC format was that a distinguished argument \( x \) of a term \( \sigma(x, \tilde{x}) \) was to be evaluated, and effects accumulated, until termination; and then execution would proceed depending on the terminal value produced. Alternatively, the behaviour of a term \( \rho(\tilde{x}) \) might be independent of the behaviour of its arguments. We respectively referred to such terms as context and redex terms.

We now express this idea categorically, by assuming additional structure on the syntax functor \( \Sigma \). We express redex constructors by an arbitrary syntax functor \( R \), and context-term construc-
tors by a polynomial bifunctor $H(X, X)$, where the first $X$ provides the ‘active’ arguments of the terms constructed, and the other $X$ is used to provide their remaining arguments; we will shorten this to $H_2X$, giving the functor $H_2X = H(X, X)$. In defining the functor $H$, we assume we are given a syntax signature $\Sigma_\text{c}$ for context terms; we adopt the shorthand $\sigma : s_0, (s_i) \to s_\sigma$ to represent the type of an operational symbol $\sigma$, where $s_0$ is the type of the active argument, and $(s_i)$ is shorthand for a sequence $(s_i)_{1 \leq i < \text{car}(\sigma)}$ of context arguments, which may be empty, written $\epsilon$. We also make use of shorthand $\prod_i$, for $\prod_{1 \leq i < \text{car}(\sigma)}$, leaving the scope of $i$ implicit. (Note that for context-term constructors, $\text{ar}(\sigma) \geq 1$).

**Definition 4.2.1.** Given a set of sorts $S$, let $C$ be a symmetric $\otimes$-monoidal category with coproducts. Let $R$ be an endofunctor on $C^S$, and $\Sigma_\text{c}$ an $S$-sorted signature for context terms. Then an endofunctor $\Sigma$ is said to be a Redex-Context (R-C) functor with respect to $R$ and $\Sigma_\text{c}$ if $\Sigma X = \text{RX} + H_2X$ where the functor $H_2 : C \to C$ is defined by $H_2X = H(X, X)$ – and the $s$-component of $H : C^2 \to C$ is defined in terms of $\Sigma_\text{c}$, as follows. (We omit the product `$\otimes \prod_i \cdots$' if there are no context arguments, i.e. $(s_i) = \epsilon$)

$$(H(X, Y))_s = \prod_{\sigma : s_0, (s_i) \to s} \left( X_{s_0} \otimes \prod_i Y_{s_i} \right)$$

**Example 4.2.2.** We may obtain an R-C syntax functor $\Sigma$ from a signature $\Sigma$ and dependency function $\text{dep}$, provided $\text{dep}(\sigma) \leq 1$ for all constructors $\sigma$. We define $\Sigma_\text{c} = \{ \sigma \in \Sigma \setminus \text{dep}(\sigma) = 0 \}$ and $\Sigma_\text{c} = \{ \sigma \in \Sigma \setminus \text{dep}(\sigma) = 1 \}$; these signatures, and the dependency function $\text{dep}$, induce syntax functors which we take to be $R$ and $H$ respectively.

$$(\text{RX})_s = \prod_{\sigma : \Sigma_\text{c}, (s_i) \to s} \left( \prod_{0 \leq i < \text{car}(f)} X_{s_i} \right) \quad \quad (H(X, Y))_s = \prod_{\sigma : \Sigma_\text{c}, s_0, (s_i) \to s} \left( X_{s_0} \otimes \prod_{1 \leq i < \text{car}(f)} Y_{s_i} \right)$$

**Example 4.2.3.** Assume a collection $X$ of $(N, E, P)$-sorted variables, and consider the fragment of **While** given by variable lookups $l$ in a set $L$, **while** loops, variable updates $l = x_n$, and **if** statements. The first two are redex constructors, which we represent by nullary terms $l$ (of numeric sort) for every element of $L$, and tuples $\langle x_e, x_p \rangle$ of arguments in $X_E \otimes X_P$. Hence, for this fragment of **While**, we can represent a single application of the redex constructors to the variables $X$ by a functor $R : \text{Cpo}^3_{\perp} \to \text{Cpo}^3_{\perp}$, given by $R(X_N, X_E, X_P) = (L_{\perp}, 0, X_E \otimes X_P)$. (This fragment of the language does not construct any boolean-sort commands, which is why the second component of $RX$ is empty.)

The remaining constructors are context-terms: we describe them in terms of a signature $\Sigma_\text{c}$ as shown below, using the notation $\sigma(\cdots) : s_0, (s_i) \to s_\sigma$ we have introduced (but including sample arguments $x_n$, $x_e$, etc.). We have variable updates for each $l \in L$, with an active argument of numeric sort $N$, and an empty collection of context arguments $\epsilon$. The **if** statement has an active argument of boolean sort $E$, and two context arguments, both of command sort $P$. 

4.2. Evaluation-In-Context Congruence (EIC) formats for Effectful and Stateful Languages

\[ l = (x_n) : N, \epsilon \rightarrow P \quad \text{if } (x_e) \text{then } \{x_p\} \text{ else } \{x'_p\} : E, (P, P) \rightarrow P \]

The resulting definition of \( H \) is shown below, where the coproduct \( \coprod \) describes a variable update for each location \( l \) – omitting an empty smash in the first coproduct, as there are no context arguments – and the last component constructs if statements, containing active arguments drawn from \( X_E \) and pairs of context arguments drawn from \( Y_P \). (We only construct command-sort terms, which is why the other components of \( H(X, Y) \) are empty.)

\[
(H(X, Y))_P = (\coprod_{l \in L} X_N) + (X_E \otimes (Y_P \times Y_P)) \quad (H(X, Y))_E = (H(X, Y))_N = 0.
\]

4.2.1 Evaluation-in-Context for cts, cets, and ets: EIC 1-3

Evaluation-in-Context for the cts: EIC1

The operational rules for While generalise to our first rule-format for specifying operational models as cts’s. Given a pair \( \langle \sigma(x, \tilde{x}), c \rangle \) of a term \( \sigma(x, \tilde{x}) \) and a comodel-state \( c \), the format requires the term \( \sigma(x, \tilde{x}) \) to imitate the behaviour of the active argument \( x \) when it is paired with the same state \( c \). As before, in an ordered setting it is convenient to consider divergence as a special return value \( \bot \).

**Definition 4.2.4.** In a concrete category \( C \), suppose we are given objects of syntax variables \( X \) and values \( V \), and a comodel \( C : L^{op} \rightarrow C \) for a Lawvere theory \( L \) in \( C \). A Concrete Evaluation-In-Context 1 (EIC1) Specification consists of the following, where for each rule below we assume \( \tilde{x} = (x_i)_{i \in I}, \tilde{y}_{v,c} = (y_j)_{j \in J_{v,c}}, \) and \( \tilde{z}_c = (z_k)_{k \in K_c} \), are such that: (1) \( \{x_i : i \in I\} \subseteq X \) are pairwise distinct and disjoint from \( \{x, x'\} \); (2) \( \{y_j : j \in J_{v,c}\} \subseteq \{x_i : i \in I\} \); and similarly (3) \( \{z_k : k \in K_c\} \subseteq \{x_i : i \in I\} \). Below, the behaviours \( b_{v,c}(\tilde{y}_{v,c}) \) and \( b_c(\tilde{z}_c) \) stand for either: (1) syntax terms \( t_{v,c}(\tilde{y}_{v,c}) \) and \( t_c(\tilde{z}_c) \) respectively; or (2) terminal values \( u \) (i.e. with no dependence on the arguments \( \tilde{y}_{v,c} \) or \( \tilde{z}_c \)).

- For every context-term constructor \( \sigma \), we require the left-hand rule (CTXL) below (parametric in \( s, s' \)), and one instance of the right-hand rule (CTXR) for every \( v \in V \) and comodel state \( c \in C_{1} \), with corresponding terminal value or term \( b_{v,c}(\tilde{y}_{v,c}) \) and new comodel state \( c'_{v,c} \). In the case that \( v = \bot \), we require \( b_{v,c} = \bot \).

\[
\frac{\langle x, s \rangle \rightarrow \langle x', s' \rangle}{\langle \sigma(x, \tilde{x}), s \rangle \rightarrow \langle \sigma(x', \tilde{x}), s' \rangle} \quad \text{(CTXL)} \]

\[
\frac{\langle x, s \rangle \rightarrow \langle v, c \rangle}{\langle \sigma(x, \tilde{x}), s \rangle \rightarrow \langle b_{v,c}(\tilde{y}_{v,c}), c'_{v,c} \rangle} \quad \text{(CTXR)}
\]
– For redex constructors \( \rho \), a rule \( \langle \rho(\hat{x}), c \rangle \rightarrow \langle b_c(\tilde{z}_c), c'_c \rangle \) (REDX) for each comodel state \( c \in C_1 \), with terminal value or term \( b_c(\tilde{z}_c) \) and new comodel state \( c'_c \).

**Example 4.2.5.** For **While** programs, consider the five operational rules given above, for sequential composition \( p; q \), variable update \( x = u \), and **while** statements (with syntax variables \( P = \{p, q, u, e\} \)). The two rules for sequential composition \( \sigma(x, \hat{x}) = p; q \) have \( x = p \), and \( \hat{x} = q \).

The first rule is simply (CTXL), and the second corresponds to (identical) instances of (CTXR) for all \( c \in C_1 \) (there is only one return value \( v = \ast \) for commands), where each rule instance has \( b_{v,c}(\tilde{x}_{v,c}) = q \), and as the comodel state \( c \) is unchanged in the transition, we take \( c'_c = c \).

The rules for variable update (where \( \sigma(x, \hat{x}) \) is the update \( l = u \)) take \( x \) to be \( u \), and \( \hat{x} \) is empty. The first rule for variable update is again (CTXL), and the second corresponds to rules (CTXR) for every \( n \in \mathbb{N} \), where \( v = n \), \( b_{v,c}(\tilde{x}_{v,c}) = \ast \), \( c'_c = c[x \mapsto n] \).

Lastly, the rule for **while** statements corresponds to (identical) instances of (REDX) for all \( c \in C_1 \), with \( \sigma(\hat{x}) = \text{while}(e)\text{do}'p' \), \( b_c(\tilde{y}_c) = \text{if}(e)\text{then}'p'\text{while}(e)\text{do}'p'\text{else}'\text{skip}' \), and \( c'_c = c \).

We now consider how an EIC1 specification may be expressed categorically, beginning with redex rules (REDX). We have a rule for every redex constructor \( \rho(\hat{x}) \) applied to syntax variables \( X \) – i.e. for each element of \( RX \) – and for each initial comodel-state \( c \in C_1 \). Hence, we may index the collection of rules by a suitable product of \( RX \) and \( C_1 \); and to exploit the \( \otimes \)-closed structure of \( C \), we use the monoidal product \( \otimes \) – so the collection is described by \( RX \otimes C_1 \).

Each such rule has to specify a behaviour (called \( t_c(\tilde{z}_c) \) above): either a terminal value in \( V \), or a new term in \( TX \) – and this is represented by the coproduct \( V + TX \), which is \( BTX \). Each rule must also give a new comodel-state \( c'_c \in C_1 \). Again, we represent this data by the monoidal product \( BTX \otimes C_1 \). Hence, the rules (REDX) of an EIC1 specification may be collected into a single arrow \( \alpha_X : (RX \otimes C_1) \rightarrow (BTX \otimes C_1) \). To allow relabelling of the variables \( X \) occurring in the rule, we assume \( \alpha \) is a natural transformation.

We now consider the rules for context terms \( \sigma(x, \hat{x}) \). Firstly, (CTXL) is generic and contains no information specific to any syntax constructor, so we need not describe it. The rules (CTXR) specify, for each terminal value \( g \) that \( x \) can take, what the behaviour of a context term \( \sigma(x, \hat{x}) \) will be if paired with a comodel-state \( c \in C_1 \). The rules are parametric in the active argument \( x \); it does not appear in the derived transition behaviour for \( \sigma(x, \hat{x}) \). Hence, the rules (CTXR) may be thought of as assigning a behaviour to each configuration \( \langle \sigma(y, \hat{x}), c \rangle \); these may be collectively represented by elements of \( H(V, X) \otimes C_1 \). To each configuration, the rules (CTXR) assign a behaviour \( t_{v,c}(\tilde{y}_{v,c}) \) which is either a terminal value \( V \) or a new term \( TX \), again represented by \( BTX \). In addition, the rules must specify a new comodel state \( c'_{v,c} \) in \( C_1 \).

Allowing for variable relabelling, this means we may represent the rules (CTXR) by a natural
transformation \( \kappa_X : (H(V,X) \otimes C1) \rightarrow (BTX \otimes C1) \). As for divergent behaviour, in the context of \( \mathsf{Cpo}_{\perp} \), the monoidal product in the domain of \( \kappa_X \) ensures that if the active argument diverges (so that \( v = \bot \)), then \( \kappa_X \) returns a divergent behaviour \( \bot \) as required (as an arrow in \( \mathsf{Cpo}_{\perp} \), it must be \( \perp \)-preserving).

This leads to the following categorical description of the EIC1 specification data:

**Definition 4.2.6.** Let \( C \) be a symmetric \( \otimes \)-monoidal closed category with (co)products, and \( \Sigma \) an R-C functor with respect to \( R \) and \( \mathsf{Sig}_r \). With respect to a comodel \( C : \mathcal{L}^{op} \rightarrow \mathcal{C} \) and an object of values \( V \), an **Abstract EIC1 Specification** consists of two natural transformations

\[
\alpha_X : (RX \otimes C1) \rightarrow (BTX \otimes C1) \quad \text{and} \quad \kappa_X : (H(V,X) \otimes C1) \rightarrow (BTX \otimes C1).
\]

**Example 4.2.7.** Consider the fragment of **While** given by if statements, variable lookups \( l \) in a set \( L \) and updates \( l = x \), where we take the standard comodel in \( \mathsf{Cpo}_{\perp} \) with \( C1 = (\mathbb{N}^L)_\perp \); we may represent these by the following syntax functors (see Example 4.2.3).

\[
R(X_N, X_E, X_P) = (L_\perp, 0, 0) \quad \text{and} \quad H(X, Y) = (0, 0, \bigoplus_{l \in L} X_N + X_E \otimes (Y_P)^2)
\]

However, it is more helpful to express the coproduct \( \bigoplus_{l \in L} \) isomorphically as an \( L \)-fold copower in \( \mathsf{Cpo}_{\perp} \), viz. \( L \cdot X_N \).

Hence, an abstract EIC1 specification is given by natural transformations

\[
\alpha_X : ((L_\perp, 0, 0) \otimes C1) \rightarrow (BTX \otimes C1) \quad \text{and} \quad \kappa_X : (0, 0, (L \cdot V_N + V_E \otimes X^p_2) \otimes C1) \rightarrow (BTX \otimes C1),
\]

where \( V_N \) is the flat cppo \( \mathbb{N}_\perp \) of natural return values, and \( V_E = \mathbb{B}_\perp \) is the flat cppo of boolean return values (where we represent the elements of \( \mathbb{B} = \{\text{false}, \text{true}\} \) by the left and right components of the two-element set \( 1 + 1 \)). Here, the monoidal product with \( C1 \) is given componentwise, as in remark 4.1.6; to separate the \( P \)-component of \( \kappa_X \), we use a distributivity isomorphism

\[
\mathsf{dist} : ((L \cdot V_N + V_E \otimes X^p_2) \otimes C1) \cong (L \cdot V_N) \otimes C1 + (V_E \otimes X^p_2) \otimes C1
\]

and then use the following isomorphisms (in \( \mathsf{Cpo}_{\perp} \))

\[
(0 \otimes C1) \cong 0 \quad L \cdot \mathbb{N}_\perp \cong (L \times \mathbb{N})_\perp \quad A_\perp \otimes C1 \cong A \cdot C1 \text{ for any set } A
\]

to isomorphically replace the domains of \( \alpha_X \) and \( \kappa_X \) with \( (L \cdot C1, 0, 0) \) and \( (0, 0, (L \times \mathbb{N}) \cdot C1 + (1 + 1) \cdot X^p_2) \cdot C1 \) respectively.

In the setting of \( \mathsf{Cpo}_{\perp}^3 \), note that a natural transformation \( \delta_X : FX \rightarrow GX \) is equivalent to three natural transformations \( (\delta_X)_s : (FX)_s \rightarrow (GX)_s \) between the component functors \( (F)_s \) and
Thus, we define $\alpha$ and $\kappa$ componentwise as follows. We trivially take the $N$-components to be the natural transformation $z_X : 0 \to (BTX \otimes C1)_N$ given by the unique arrows from the initial object 0 – and similarly for the E components. We deal with the $P$-components by defining natural transformations $(\alpha_P)_X : L \cdot C1 \to (BTX)_N \otimes C1$ describing lookups, and take $(\kappa_P)_X$ to be the coproduct $\beta_X + \gamma_X$ of a natural transformation $\beta_X : (L \times V) \cdot C1 \to ((BTX)_P \otimes C1)$ specifying updates, and $\gamma_X : ((1 + 1) \cdot X^2_P) \otimes C1 \to (BTX)_P \otimes C1$ specifying if statements.

We define the specifications $\alpha_X$ and $\beta_X$ of lookups and updates by making use of the comodel implementations of lookups $lku : C1 \to N \cdot C1$ on locations $l$, and updates $upd_{l,n} : C1 \to C1$, as described in Section 2.3.3. Taking $L$-fold coproducts of lookups $lku$ for each location, we obtain the first arrow below, which returns the value $n$ of $l$ and the comodel-state $C1$; we use some isomorphisms and an injection to construe $n$ as a return value in $BTX$, as required.

$$(\alpha_P)_X : L \cdot C1 \xrightarrow{[lku]} N \cdot C1 \cong N_\perp \otimes C1 \xrightarrow{[\text{inl}]_\perp \otimes C1} (N_\perp \oplus (TX)_N) \otimes C1 = (BTX)_N \otimes C1$$

Similarly, we specify updates via an $(L \times N)$-fold coproduct of the maps $\text{upd}_{l,n}$, a monoidal isomorphism (essentially introducing the ‘void’ return value $\bot$, in $V_P = 1_\perp$), and an injection into $(BTX)_P$.

$$\beta_X : (L \times N) \cdot C1 \xrightarrow{[\text{upd}_{l,n}]} C1 \cong 1_\perp \otimes C1 \xrightarrow{[\text{inl}]_\perp \otimes C1} (1_\perp \oplus (TX)_P) \otimes C1 = (BTX)_P \otimes C1$$

Lastly, we specify if $\{p\} \text{ else } \{q\}$ statements, which essentially do not interact with the comodel, by using suitable projections to isolate the appropriate successors $p, q$, as follows:

$$\gamma_X : ((1 + 1) \cdot X^2_P) \otimes C1 \cong (X^2_P + X^2_P) \otimes C1 \xrightarrow{([\pi_2 + \pi_1] \otimes \text{id})} X_P \otimes C1 \xrightarrow{(\text{inl}(TX)_P \otimes \text{id})} (BTX)_P \otimes C1$$

**Evaluation-in-Context for the ets: EIC2**

In Section 3.4.1, the concept of evaluation-in-context was used to define the eEIC rule format for languages with syntactic effects, considered as syntactic ETS’s; it involved an effectful extension of a restricted class of effect-free abstract OS specifications. In this section, we give a direct and more general formulation of this idea, called the EIC2 format, without recourse to effectful extensions; it is made categorical in a similar way to the EIC1 format for CTS’s. The resulting format is syntactically more complicated, but gives more insight into the mechanics of operational models for effectful languages, and generalises to a rule format for the CETS.

We begin by illustrating how the operational rules for variable lookup and update in While would appear in this format, before giving the formal definition and explanation. Once again, we treat divergence $\bot$ as a special return value.
\[
l \rightarrow \text{rd}_i(0, 1, 2, \ldots)
\]

\[
x = u \rightarrow \delta \left( \begin{array}{ll}
x = u' & \text{if } b_k = u' \\
\text{wr}_{x,n}(\tilde{x}) & \text{if } b_k = n
\end{array} \right)_{k \in K}
\]

**Definition 4.2.8.** In a category \(C\), suppose we are given objects of syntax variables \(X\) and values \(V\), and a signature \(\text{Eff}\) of syntactic effects drawn from a theory \(\mathcal{L}\). A **Concrete Evaluation-In-Context 2 (EIC2) Specification** consists of the following, where we make analogous assumptions to Definition 4.2.4 on \(\tilde{x}, \tilde{y},\) and \(\tilde{z}\). Again, the behaviours \(b_l(\tilde{y}), b_l(\tilde{z})\) are either syntax terms \(t_l(\tilde{y}), t_l(\tilde{z})\), or terminal values \(\nu_l\).

- For each redex constructor \(\rho\), we require a rule \(\rho(\tilde{x}) \rightarrow \epsilon((b_l(\tilde{z}))_{l \in L})\) (REDX), with a syntactic effect-tree \(\epsilon\) whose \(L\)-indexed leaves \(b_l(\tilde{z})\) are either terminal values \(\nu_l\), or new terms \(t_l\).

- For every context-term constructor \(\sigma\), we require an instance of the rule (CTXB) shown below, for every effect-tree \(\delta\) with leaves \(\{b_k : k \in K\}\) given by either a syntax variable \(x_k\) or a terminal value \(\nu_k\). (Essentially, we are asserting that the rule below is parametric in the effect tree \(\delta(b_k)_{k \in K}\).) We assume these \(x_k\) are all distinct, and do not include \(x\) or any arguments in \(\tilde{x}\). Across all of these instances, we assume a common choice of \(V\)-indexed effect-trees \(\epsilon(\nu_l(b_l(\tilde{y}))_{l \in L_v})\) with \(L_v\)-indexed leaves \(b_l(\tilde{y})\) given by terminal values \(\nu_l\) or terms \(t_l(\tilde{y})\). We require that \(\epsilon_\bot\) is the trivial tree with a single leaf \(\bot\).

\[
\sigma(x, \tilde{x}) \rightarrow \delta \left( \begin{array}{ll}
\sigma(x_k, \tilde{x}) & \text{if } b_k = x_k \\
\epsilon_l((b_l(\tilde{y}))_{l \in L_v}) & \text{if } b_k = \nu
\end{array} \right)_{k \in K}
\]

(CTXB)

Under the EIC2 format, a redex-term \(\rho(\tilde{x})\) (such as the lookups \(l\) above) can exhibit an arbitrary effect-tree \(\epsilon((b_l(\tilde{z}))_{l \in L})\) of transition behaviours; the leaves \(b_l(\tilde{z})\) may be terminal values \(\nu_l\), or arbitrary new terms \(t_l(\tilde{z})\) built from the arguments in \(\tilde{x}\). For example, in the case of variable lookups \(l\) in **While**, \(\epsilon\) consists of a single read-effect, and its leaves are all terminal values \(\nu_l\).

Context-terms \(\sigma(x, \tilde{x})\) are more complex, as they behave according to the effect-tree introduced by the active argument: \(x \rightarrow \delta((b_k)_{k \in K})\). The leaves of this effect-tree are behaviours \(b_k\) given either by new states \(x'_k\), or terminal values \(\nu_k\). The term \(\sigma(x, \tilde{x})\) should then exhibit (at least) the same effects \(\delta((b'_k)_{k \in K})\) as \(x\); here, each \(b'_k\) is either a new subtree (extending \(\delta\)), or a new term, and is determined from the corresponding leaves \(b_k\) in the behaviour of \(x\), as follows. If the leaf \(b_k\) is a new state \(x'_k\), then the corresponding leaf \(b'_k\) substitutes that state for the active argument \(x\), giving the term \(\sigma(x'_k, \tilde{x})\). Otherwise if the leaf \(b_k\) is a terminal value \(\nu\), then \(b'_k\) is allowed to be an effect-tree \(\epsilon_l((b_l(\tilde{y}))_{l \in L_v})\) depending on this value. Its leaves are behaviours \(b_l(\tilde{y})\): either terminal values \(\nu_l\), or arbitrary new terms \(t_l(\tilde{y})\) built from the remaining arguments \(\tilde{x}\) of the context-term (but not the active argument). For example, in the
case of variable updates \( l = u \) for \texttt{While}, where the terminal values \( v \) for numeric expressions are \( \mathbb{N} \)-valued, each effect-tree \( \epsilon_n \) consists of a single write operation \( \texttt{wr}_{x,n}(b_1) \), and its leaf \( b_1 \) is a terminal value \( \ast \).

We now consider how the above information appears categorically, just as we did for the \texttt{cts}. The rules (\texttt{REDX}) are indexed by redex terms \( \rho(\bar{x}) \) in \( RX \), and each rule involves a syntactic effect-tree \( \epsilon((b_l(\bar{z}_i))_{i \in L}) \) with leaves given by transition behaviours: either terminal values \( v_j \) in \( V \), or terms \( t_l(\bar{z}_i) \) in \( TX \); we combine both possibilities into the coproduct \( BTX = V + TX \), and the syntactic effect-trees over these transitions are represented by \( T_e BTX \). Thus, we represent the rules (\texttt{REDX}) by a natural transformation \( \alpha_X : RX \to T_e BTX \).

We now consider the rules (\texttt{CTXB}) for context terms \( \sigma(x, \bar{x}) \). Assuming the active argument \( x \) has behaviour \( x \to \delta((b_k)_{k \in K}) \), the rules assign a behaviour to \( \sigma(x, \bar{x}) \) by examining the leaves \( b_k \). If they are non-terminal transitions to states \( x_k \), the rules replace those leaves with \( \sigma(x_k, \bar{x}) \); this is generic to the rule format. However, if they are terminal values \( v_l \), then the rules specify the effect-trees \( \epsilon_v((b_l(\bar{y}_i))_{i \in L_v}) \) that are to be substituted for those terms. As before, the effect-trees are of type \( T_e BTX \), and they cannot make any reference to the active argument \( x \), but only the other arguments \( \bar{x} \) and the terminal value \( v_l \) of the active argument; so these effect-trees are essentially allocated to each term \( \sigma(v_l, \bar{x}) \), where the active argument has been replaced by a terminal value. These terms are elements of the object \( H(V, X) \). Hence, the rules (\texttt{CTXB}) may be represented collectively by a natural transformation \( \kappa_X : H(V, X) \to T_e BTX \). This leads to the following categorical definition:

**Definition 4.2.9.** Let \( C \) be a symmetric monoidal closed category with countable (co)products, and \( \Sigma \) an \( R \)-\( C \) functor with respect to \( R, H \). With respect to an object of values \( V \), a Syntactic Abstract EIC2 Specification consists of two natural transformations \( \alpha_X : RX \to T_e BTX \) and \( \kappa_X : H(V, X) \to T_e BTX \).

**Example 4.2.10.** We illustrate how the specification for \texttt{sEWhile} may be described in these terms, in \( \text{Cpo}_{\text{L}, \ell} \). We consider the fragment of read and write effects \( \texttt{rd}_l, \texttt{wr}_{l,n} \) at every type, and \texttt{if} statements. The former are given by the single-sorted effect signature

\[
\text{Eff} = \{(\texttt{rd}_l : s^\omega \to s) : l \in L\} \cup \{(\texttt{wr}_{l,n} : s \to s) : l \in L, n \in \mathbb{N}\}
\]

which induces a single-sorted effect-syntax functor \( \Delta_0 X = \coprod_{l \in L} X^\omega + \coprod_{(l,n) \in L \times \mathbb{N}} X \), or isomorphically \( \Delta_0 X = L \cdot X^\omega + (L \times \mathbb{N}) \cdot X \). This induces the syntactic effect monad \( T_{\omega o} \), and both generalise componentwise to \( \Delta, T_e \) respectively; the syntax of the language is then given by \( \Sigma = \Sigma_0 + \Delta \), where \( \Sigma_0 \) is induced by the effect-free fragment of the language.

The specification of the above three commands amount to natural transformations, \( (\alpha_s)_X : L \cdot X_s^\omega \to (T_e BTX)_s \) for read-effects at sort \( s \), \( (\beta_X : (L \times \mathbb{N}) \cdot 1 \to (T_e BTX)_s \) for write effects, and \( (\gamma_X : (1 + 1) \cdot X_p^2 \to (T_e BTX)_p \) for \texttt{if} statements. The difference is that both read and
write effects are \textit{redexes}, with transitions such as \( \text{wr}_{x,n}(p) \to \text{wr}_{x,n}(p) \); furthermore, they are allowed to occur at every sort.

Lookups and updates are essentially specified in the same way as the maps \( \epsilon^{(1)} \) in Section 3.3.4, by construing syntactic effects as behaviour. We illustrate the definition of \( (\alpha_s)_X \); \( (\beta_s)_X \) is similar.

\[
(\alpha_s)_X : L \cdot X \omega \xrightarrow{\text{inl}_X \omega} \Delta_0 X = (\Delta X)_s \xrightarrow{\epsilon} (T_e)^s \xrightarrow{\epsilon} (T_e B)^s \xrightarrow{(\eta_T e X)} (T_e B)^s.
\]

We specify if statements in the same way as Example 4.2.7, but now using the \( (P\text{-component of the}) \) monad unit \( \eta_T e \) to describe an effect-free transition.

\[
\gamma_X : (1 + 1) \cdot X^2 \xrightarrow{\pi_2 + \pi_1} X \xrightarrow{(\text{inr}_T X^p \cdot \eta_T e X^p)_p} (BT X)_p \xrightarrow{(\eta_{BTX})^p} (T_B BTX)_p.
\]

We remark that operational specifications are inherently syntactic in nature, and it is natural to describe a rule format in terms of syntactic effects \( \epsilon(\tilde{b}) \) (and the monad \( T_e \)), rather than their semantic equivalence classes \( [\epsilon(\tilde{b})] \) (i.e. the monad \( N_e \)). Generally, these equivalence classes do not lend themselves to a simple syntactic presentation; for instance, if the effects are given by the theory of global store, these equivalence classes may be characterised as functions in \((S \cdot X)^S\) (by Example 4.1.3), and we arrive at an unconventional notion of ‘operational rules’ resembling that discussed in Section 3.1.2.

However, on a categorical level, it is easy to adapt the abstract EIC2 format simply by replacing the monad \( T_e \) with \( N_e \): this amounts to a description of program behaviour in terms of equivalence classes of effect-trees.

\textbf{Definition 4.2.11.} Under the same assumptions as Definition 4.2.6, with respect to an object of values \( V \), a \textit{Semantic Abstract EIC2 Specification} consists of two natural transformations \( \alpha_X : RX \to N_e BTX \) and \( \kappa_X : H(V, X) \to N_e BTX \).

Indeed, we may convert a syntactic abstract EIC2 specification into a semantic one, simply by post-composing the natural transformations \( \alpha_X : RX \to T_e BTX \), \( \kappa_X : H(V, X) \to T_e BTX \) with the monad morphism \( \text{quot}_{BT} : T_e BT \Rightarrow N_e BT \) which sends syntactic effect-trees to their equivalence classes. This removes the distinction between syntactic effect-trees which should be considered equivalent under the algebraic theory for the effects; we use this idea to define a behavioural equivalence \( \cong^e_{BT} \) for syntactic ETS’s which achieves this in Section 4.3.2.

\textbf{Evaluation-in-Context for the cets: EIC3}

Our final rule format specifies operational models for languages combining comodels and syntactic or semantic effects, as cets’s. As before, we assume the effects come from a Lawvere
theory $\mathcal{L}_2$, and that the comodel is for a theory $\mathcal{L}_1$.

We give example rules for binary choice or and assignments $x = u$ in NDWhile, where $\delta(\ldots)$ stands for an arbitrary syntactic or-tree of pairs $\langle b_k, c_k \rangle$, in which $b_k$ is either a terminal value $v$ or a program state $u'$, and $c_k$ a comodel-state; the general format follows.

$$
\langle x \ or\ y, s \rangle \rightarrow \langle x, s \rangle \ or\ \langle y, s \rangle
$$

$$
\langle x = u, s \rangle \rightarrow \delta \left( \begin{array}{l}
\langle x = u', c_k \rangle \quad \text{if } b_k = u' \\
\langle \ast, c_k[x \mapsto n] \rangle \quad \text{if } b_k = v
\end{array} \right)_{k \in K}
$$

**Definition 4.2.12.** In a category $\mathcal{C}$, suppose we are given objects of syntax variables $X$ and values $V$, a comodel $C : \mathcal{L}_2^p \rightarrow \mathcal{C}$, and a signature $\text{Eff}$ of syntactic effects drawn from a theory $\mathcal{L}_2$. A Concrete Evaluation-In-Context 3 (EIC3) Specification consists of the following, where we make analogous assumptions to Definition 4.2.4 on $\tilde{x}, \tilde{y}_l$, and $\tilde{z}_i$. Again, the behaviours $b_l(\tilde{y}_l), b_l(\tilde{z}_i)$ are either syntax terms $t_l(\tilde{y}_l), t_l(\tilde{z}_i)$, or terminal values $\varepsilon_l$.

- For redex constructors $\rho$, we require a rule $\langle \rho(\tilde{x}), c \rangle \rightarrow \varepsilon_c(\langle b_l(\tilde{z}_i), c'_l \rangle)_{l \in L_2}$ (REDX) for all comodel states $c$, with syntactic effect-trees $\varepsilon_c$ whose $L_2$-indexed leaves $\langle b_l(\tilde{z}_i), c'_l \rangle$ are pairs of a behaviour $b_l(\tilde{z}_i)$ and a new comodel state $c'_l$.

- For every context-term constructor $\sigma$, we require an instance of the rule (CTXB) below, for every effect-tree $\delta$ with leaves $\langle b_k, c_k : k \in K \rangle$ given by pairs $\langle b_k, c_k \rangle$ of: either a syntax variable $x_k$ or a terminal value $v_k$; and a comodel-state $c_k$. We assume these $x_k$ are all distinct, and do not include $x$ or $x_i$ for $i \in I$. Across all these rule-instances, we assume a common choice of effect-trees $\varepsilon_{x_i, c}$ indexed by pairs of terminal values $v$ and comodel-states $c$; and their $L_v, c_k$-indexed leaves $\langle b_i(\tilde{y}_l), c'_l \rangle$ are given by pairs of behaviours $b_l(\tilde{y}_l)$ and comodel-states $c'_l$, as above.

$$
\langle \sigma(x, \tilde{x}), s \rangle \rightarrow \delta \left( \begin{array}{c}
\langle \sigma(x_k, \tilde{x}), c_k \rangle \\
\varepsilon_c(\langle b_l(\tilde{y}_l), c'_l \rangle)_{l \in L_2, c_k}
\end{array} \right)_{k \in K}
$$

(CTXB)

To express an EIC3 specification categorically, note that the rules (REDX) are again indexed by the redex terms $\rho(\tilde{x})$ over $X$ and by the initial comodel state $c$, just as for the EIC1 format; hence we index the collection by $RX \otimes C1$ again. Each rule conclusion now contains a syntactic effect-tree $\varepsilon_c$ over pairs consisting of a behaviour $b_l(\tilde{z}_i)$ – either a terminal value $V$ or a term $t_l(\tilde{z}_i)$ – and a new comodel state $C1$; and this effect-tree is represented by $T_0^{(2)}(BTX \otimes C1)$, where $T_0^{(2)}$ is the syntactic effect-monad given by a signature $\text{Eff}$ of effects drawn from $\mathcal{L}_2$. Hence, the rules (REDX) are represented by a natural transformation $\alpha_X : (RX \otimes C1) \rightarrow T_0^{(2)}(BTX \otimes C1)$.

The rules (CTXB) for context terms $\sigma(x, \tilde{x})$ are formalised in close analogy to EIC2. Under the assumption that $\langle x, c \rangle \rightarrow \delta(\langle b_k, c_k \rangle)_{k \in K}$, the rules assign a behaviour to $\langle \sigma(x, \tilde{x}), c \rangle$ by examining
the leaves \( b_k \). If they are states \( x'_k \), the rules generically replace them with \( \langle \sigma(x'_k, \hat{x}), c_k \rangle \). If they are terminal values \( y \), the rules specify effect-trees \( e_{v,c_k}(\langle b_l(y), c'_l \rangle)_{l \in L_{v,c_k}} \) that are to be substituted for those terms, depending on the terminal value \( y \) and new comodel-state \( c_k \).

These effect-trees are of type \( T_e^{(2)}(BTX \otimes C1) \), and they can only depend on the non-active arguments \( \hat{x} \) and terminal value \( y \) of the active argument \( x \); so they are essentially allocated to configurations \( \langle \sigma(y, \hat{x}), c \rangle \), which are elements of the object \( H(V, X) \otimes C1 \). Hence, the rules (CTXB) may be represented collectively by a natural transformation \( \kappa_X : (H(V, X) \otimes C1) \rightarrow T_e^{(2)}(BTX \otimes C1) \).

**Definition 4.2.13.** Under the same assumptions as Definition 4.2.6, with respect to an object of values \( V \) and a comodel \( C : L_2^{op} \rightarrow C \), a **Syntactic Abstract EIC3 Specification** consists of two natural transformations \( \alpha_X : (RX \otimes C1) \rightarrow T_e^{(2)}(BTX \otimes C1) \) and \( \kappa_X : (H(V, X) \otimes C1) \rightarrow T_e^{(2)}(BTX \otimes C1) \).

**Example 4.2.14.** We consider the **NDWhile** language, with binary choice \( \text{or} \), and corresponding effect-syntactic functor \( \Delta_0 X = X^2 \). The deterministic fragment of the language is given by the constructors of **While** language; and we may trivially adapt an abstract EIC1 specification for **While**, as illustrated by Example 4.2.7, into an EIC3 specification for this fragment of **NDWhile**, by post-composing with the unit \( \eta^{T_e^{(2)}} \) of the syntactic effect monad \( T_e^{(2)} \). To illustrate, given an EIC1 specification \( \alpha', \kappa' \) of redex and context-term constructors in **While**, we obtain EIC3 specifications as follows.

\[
\begin{align*}
\alpha_X : \ (RX \otimes C1) & \xrightarrow{\alpha'_X} \ (BTX \otimes C1) \xrightarrow{\eta^{T_e^{(2)}}} \ T_e^{(2)}(BTX \otimes C1) \\
\kappa_X : \ (H(V, X) \otimes C1) & \xrightarrow{\kappa'_X} \ (BTX \otimes C1) \xrightarrow{\eta^{T_e^{(2)}}} \ T_e^{(2)}(BTX \otimes C1)
\end{align*}
\]

The specification of the syntactic effect \( \text{or} \), at each sort \( s \), is given as follows:

\[
(\gamma_s)_X : \ (X_s^2 \otimes C1) \xrightarrow{(\text{inl}_{X_s} \otimes \text{id})_s} \ \Delta_0 X_s \otimes C1 = (\Delta X \otimes C1)_s \xrightarrow{((\Delta \eta^{T_e^{(2)}}(X \otimes C1))_s^{\otimes \text{id}})} \ (\Delta T_e^{(2)} X \otimes C1)_s \xrightarrow{((T_e^{(2)} \text{inr}_{X} \otimes \text{id})_s)} \ (T_e^{(2)}(BX \otimes C1))_s \xrightarrow{(T_e^{(2)}Bj_X \otimes \text{id})_s} \ (T_e^{(2)}BTX \otimes C1)_s.
\]

As discussed for the EIC2 format, the above definition has been with respect to syntactic effects, drawn from the theory \( L_2 \); and there is an analogous definition in terms of semantic equivalence classes, which is once again of little practical use, but will play a part in reasoning about a behavioural equivalence for syntactic cets’s.

**Definition 4.2.15.** Under the same assumptions as Definition 4.2.13, a **Semantic Abstract EIC3 Specification** consists of natural transformations \( \alpha_X : (RX \otimes C1) \rightarrow N_e^{(2)}(BTX \otimes C1) \) and \( \kappa_X : (H(V, X) \otimes C1) \rightarrow N_e^{(2)}(BTX \otimes C1) \).
As before, we may convert a syntactic abstract EIC3 specification into a semantic one by post-composing with the monad morphism quot_{BT(-) \otimes C1}, which has the desirable effect of identifying semantically equivalent effect-trees; and this is used to define a behavioural equivalence ≃_{Nce} for syntactic cets’s in Section 4.3.3.

4.2.2 Unifying the Formats: The Abstract EIC Format

Having introduced categorical descriptions of the evaluation-in-context rule format for the three classes of transition systems, we show that they may be considered as instances of a single EIC format for different monads \(M\) on the \(S\)-fold product \(C^S\) of a symmetric monoidal category \(C\), and express this monadic format as an abstract OS specification of the form introduced by Turi and Plotkin. As before, we assume the monads are defined identically componentwise\(^1\), so that \((MX)s = M_0(Xs)\) for some monad \(M_0\) on \(C\); for shorthand, we say \(M\) is a componentwise monad. We assume that \(M_0\), and hence \(M\), have strengths with respect to the monoidal product \(\otimes\) (for instance, by \(C\)-enrichment, as described in Section 2.1.4).

**Definition 4.2.16.** Let \(C\) be a symmetric \(\otimes\)-monoidal closed category, and \(S\) a set of sorts. On \(C^S\), let \(B\) be an endofunctor \(BX = V + X\), \(\Sigma\) an \(R\)-\(C\) syntax functor with respect to \(R,H,\) and \(M\) a monad given componentwise by a \(\otimes\)-strong monad \(M_0\) on \(C\). Then a Monadic EIC Specification consists of natural transformations \(r_X : RX \to MBTX\) and \(e_X : H(V,X) \to MBTX\).

In the case of the syntactic or semantic cets, the Abstract EIC2 specifications are immediately seen to be monadic by taking \(M = T_{ce}, N_e\) respectively. Given an Abstract EIC1 specification for a cts, one obtains a monadic EIC specification with respect to \(N_e\) by using the monoidal closed structure of the category \(C\), re-expressing the natural transformation \(\alpha_X : (RX \otimes C1) \to (BTX \otimes C1)\) in the form \(r_X : RX \to (BTX \otimes C1)^{C1} = N_e BTX\), and similarly obtaining \(e_X : H(V,X) \to (BTX \otimes C1)^{C1}\). Abstract EIC3 specifications for the syntactic or semantic cets are similarly manipulated into monadic specifications with respect to \(T_{ce}\) and \(N_{ce}\) respectively; for instance, in the syntactic case one obtains natural transformations \(r_X : RX \to (T_{e}^{(2)}(BTX \otimes C1))^{C1} = T_{ce} BTX\) and \(e_X : H(V,X) \to (T_{e}^{(2)}(BTX \otimes C1))^{C1} = T_{ce} BTX\).

We now show how a monadic EIC specification may be used to define an abstract OS specification, along similar methods to the effectful extensions of Section 3.3.4. First, it is convenient to generalise the monadic costrength of \(M_0\), viz. \(\text{cost}_{X,Y}^{(0)} : M_0X \otimes Y \to M_0(X \otimes Y)\), which played the part of ‘propagating effects’ in Section 3.3.4. Here, we propagate effects and/or comodel-manipulations from the active argument \(Xs_0\) of context terms \(H(X,Y)\), via the natural transformation \(\text{cost}' : H(MX,Y) \to MHH(X,Y)\) defined as follows.

\(^1\)We make this assumption for convenience, but one may relax the assumption that each component is identically defined; in fact, every monad on \(C^S\) is equivalent to a set of monads \(M^{(s)}\) for each sort \(s\).
Lemma 4.2.17. The following diagrams commute.

\[
\begin{align*}
(H(MX, Y))_{s} & = \coprod_{\sigma : \Sigma_{0}} (MX)_{\sigma} \otimes \prod_{i} Y_{i_{\sigma}} \\
& = \coprod_{\sigma : \Sigma_{0}} M_{0}(X_{\sigma}) \otimes \prod_{i} Y_{i_{\sigma}}
\end{align*}
\]

Here, we have omitted the scope \(1 \leq i < \text{ar}(\sigma)\) in \(\prod_{i} Y_{i_{\sigma}}\), and abbreviate \(\text{cost}^{(0)}_{X_{\sigma} \otimes \prod_{i} Y_{i_{\sigma}}}\) on the third line. On the fourth line, each map \(\text{inj}_{s}\) injects the object \(X_{\sigma} \otimes \prod_{i} Y_{i_{\sigma}}\) into the \(\sigma\)-component of the coproduct defining \(H(X, Y)\); we apply \(M_{0}\) to each such injection and take their coproduct over all context-term constructors \(\sigma \in \Sigma_{0}\).

As one might expect, this generalisation has similar properties to the monadic costrength, which we will require in our proof of adequacy and compositionality.

**Proof.** Abbreviating \(\prod_{1 \leq i < \text{ar}(\sigma)} Y_{i_{\sigma}}\) to \(Z_{\sigma}\), we break down the \(s\)-components of the first diagram as follows. (Note that the monad unit \(\eta\) is given componentwise by \(\eta^{(0)}\).) The left-hand triangle is a coproduct \(\coprod_{\sigma}\) of one of the axioms for monadic costrength; and the upper-left path in the wedge is the same as the arrow in the left-hand side of the equation at the bottom, by the general property \([g_{\sigma}]_{\sigma} \circ \coprod_{\sigma} f_{\sigma} = [g_{\sigma} \circ f_{\sigma}]_{\sigma}\) of coproducts.

\[
\begin{align*}
\coprod_{\sigma} M_{0}(X_{\sigma} \otimes Z_{\sigma}) & \xrightarrow{\coprod_{\sigma} \text{cost}^{(0)}_{X_{\sigma} \otimes Z_{\sigma}}} \coprod_{\sigma} M_{0}(X_{\sigma} \otimes Z_{\sigma}) \\
& \xrightarrow{\coprod_{\sigma} [M_{0}(\text{inj}_{s})]_{\sigma}} M_{0}(\coprod_{\sigma} X_{\sigma} \otimes Z_{\sigma})
\end{align*}
\]

The equation shown on the bottom-right arrow is proven below, where the first equality is by naturality of \(\eta^{(0)}\), the second is a property of coproducts (viz. \([f \circ g_{\sigma}]_{\sigma} = f \circ [g_{\sigma}]_{\sigma}\)), and the third uses the easily-proven fact that \([\text{inj}_{s}]_{\sigma} = \text{id}\).

\[
[(M_{0}\text{inj}_{s}) \circ \eta^{(0)}_{X_{\sigma} \otimes Z_{\sigma}}]_{\sigma} = [\eta^{(0)}_{\coprod_{\sigma} X_{\sigma} \otimes Z_{\sigma}} \circ \text{inj}_{s}]_{\sigma} = [\eta^{(0)}_{\coprod_{\sigma} X_{\sigma} \otimes Z_{\sigma}}]_{\sigma} = [\text{inj}_{s}]_{\sigma} = [\eta^{(0)}_{\coprod_{\sigma} X_{\sigma} \otimes Z_{\sigma}}]_{\sigma}
\]

We now consider the \(s\)-component of the second diagram, whose domain is a coproduct \(\coprod_{\sigma}(\cdots)\). We restrict attention to the \(\sigma\)-component of this coproduct and prove the following diagram commutes; by taking the coproduct over all \(\sigma\) of the outer paths, we obtain the \(s\)-component.
of the second diagram. The left-most rectangle is one of the axioms of the costrength $\cost^{(0)}$ and the bottom-right square is naturality of $\mu^{(0)}$. The right-most curved wedge follows again from the identity $f \circ [g\sigma] = [f \circ g\sigma]$; finally, the neighbouring area commutes by (applying $M_0$ to) the identity $[g\sigma] \circ \text{inj}_\sigma = g\sigma$, where $g\sigma = M_0(\text{inj}_\sigma) \circ \cost^{(0)}_{X^0X^0Z\sigma}$ (and the left- and right-hand sides of the identity correspond to the top-right and bottom-left paths respectively).

$$
\begin{align*}
(M_0)^2 X_0 \otimes Z_\sigma & \xrightarrow{\cost^{(0)}_{X_0X_0X_0Z_\sigma}} M_0((M_0 X_0) \otimes Z_\sigma) \\
& \xrightarrow{M_0 \text{inj}_\sigma} M_0(\bigoplus_\sigma M_0(X_0) \otimes Z_\sigma) \\
& \xrightarrow{M_0 \sqcup_{\sigma} \cost^{(0)}_{X_0X_0X_0Z_\sigma}} M_0(\bigoplus_\sigma M_0(X_0) \otimes Z_\sigma) \\
& \xrightarrow{M_0 \text{inj}_\sigma} M_0(\bigoplus_\sigma X_0) \otimes Z_\sigma
\end{align*}
$$

In addition to the adapted costrength $\cost'$, we also require an adaptation of the distributivity isomorphism $\dist'_{X,Y,Z} : H(X + Y, Z) \to H(X, Z) + H(Y, Z)$, defined as follows. The third line employs the underlying distributivity isomorphism $\dist_{P,Q,R} : (P + Q) \otimes R \to (P \otimes R) + (Q \otimes R)$, and the following line is the isomorphism $\bigoplus_\sigma (P + Q) \cong \bigoplus_\sigma (P) + \bigoplus_\sigma (Q)$.

$$(\dist'_{X,Y,Z})_\sigma : (H(X + Y, Z))_\sigma = \bigoplus_\sigma ((X + Y)_\sigma \otimes \bigoplus_i Z_{\sigma i}) = \bigoplus_\sigma ((X_\sigma + Y_\sigma) \otimes \bigoplus_i Z_{\sigma i}) \cong \bigoplus_\sigma (X_\sigma \otimes \bigoplus_i Z_{\sigma i}) + \bigoplus_\sigma (Y_\sigma \otimes \bigoplus_i Z_{\sigma i}) = H(X, Z)_\sigma + H(Y, Z)_\sigma$$

**Remark 4.2.18.** We will find it useful to note that the inverse of $\dist'_{X,Y,Z}$ is given by the natural transformation

$$[H(\text{inl}_X, \text{id}), H(\text{inr}_Y, \text{id})] : H(X, Z) + H(Y, Z) \to H(X + Y, Z)$$

or, in more detail,

$$\bigoplus_\sigma (\text{inl}_{X^0} \otimes \text{id})_\sigma \bigoplus_\sigma (\text{inr}_{Y^0} \otimes \text{id})_\sigma : \bigoplus_\sigma (X^0 \otimes \bigoplus_i Z_{\sigma i}) + \bigoplus_\sigma (Y^0 \otimes \bigoplus_i Z_{\sigma i}) \to \bigoplus_\sigma (X^0 + Y^0) \otimes \bigoplus_i Z_{\sigma i}.$$

Having adapted costrength and distributivity to deal with context terms $H(X, Y)$, we may now define the abstract OS specification induced by a monadic EIC specification.
Definition 4.2.19. Under the assumptions of Definition 4.2.16, suppose we are given a monadic EIC specification $r_X : RX \to MBTX$ and $\epsilon_X : H(V, X) \to MBTX$. The corresponding Abstract EIC Specification $\epsilon_X : \Sigma(TX \times MBTX) \to MBTX$ (with respect to monad $M$) is given by

$$\epsilon_X : R(TX \times MBTX) + H_2(TX \times MBTX) \xrightarrow{[\text{aosr}_X, \text{aosc}_X]} MBTX$$

where $\text{aosr}$ and $\text{aosc}$ are defined below. We have abbreviated $\text{cost}'_{BTX,TX}$.

$$\text{aosr}_X : R(TX \times MBTX) \xrightarrow{RTX} RTX \xrightarrow{TTX} MBTTX \xrightarrow{M\text{BT}X} MBTX$$

$$\text{aosc}_X : H_2(TX \times MBTX) \xrightarrow{H(\pi_2, \pi_1)} H(MBTCX, TX) \xrightarrow{\text{cost}'} MH(BTX, TX) \xrightarrow{M\text{dwc}_X} M^2BTCX \xrightarrow{\eta_{BTX}} MBTX$$

Here, $\text{dwc}$ (‘deal with contexts’) is defined as follows, with sub-cases handled by $\text{dwc}^{(v)}$ (‘values’) and $\text{dwc}^{(b)}$ (‘non-terminal behaviour’). We abbreviate the generalised distributivity $\text{dist}'_{V,TX,TX}$; recall that $\psi_X : \Sigma TX \to TX$ is the $\Sigma$-algebra structure of $TX$, the free $\Sigma$-algebra over $X$.

$$\text{dwc}_X : H(BTX, TX) = H(V + TX, TX) \xrightarrow{\text{dist}'_{V,TX,TX}} H(V, TX) + H_2TX \xrightarrow{[\text{dwc}^{(v)}_X, \text{dwc}^{(b)}_X]} MBTX$$

$$\text{dwc}^{(v)}_X : H(V, TX) \xrightarrow{\psi_X} MBTX$$

$$\text{dwc}^{(b)}_X : H_2TX \xrightarrow{\text{inr}} \Sigma TX \xrightarrow{\psi_X} TX \xrightarrow{\text{inr}} BTX \xrightarrow{\eta_{BTX}} MBTX$$

The map $\text{aosr}_X$ encodes the rules (REDX) occurring in all the concrete EIC formats, via their categorical description $r_X$. The rules for context terms $\sigma(t, \tilde{t})$, of type $H_2TX$, are to be applied at each branch of the behaviour $MBTX$ of the active argument (e.g. $\delta((b_i)_{i \in I})$, for a syntactic ETS). The first line of $\text{aosc}_X$ substitutes this behaviour $\delta((b_i)_{i \in I})$ for the active argument $t \in TX$, giving pairs (e.g. $\sigma(\delta((b_i)_{i \in I}), \tilde{t})$) of type $H(MBTCX, TX)$. We then use the generalised costrength $\text{cost}'$ to attach the context to each computation branch (e.g. $\delta((\sigma(b_i, \tilde{t}))_{i \in I})$).

The map $\text{dwc}_X$ then decides what to do for each computation branch $(\sigma(b_i, \tilde{t}))$. If the behaviour $b_i$ at that branch is a terminal value $v$, then we use the natural transformation $e$, from the monadic EIC specification data, to decide what the behaviour should be at that branch $(\sigma(v, \tilde{t}))$; this is handled by the map $\text{dwc}^{(v)}$, corresponding to the rules (CTXR). Otherwise, if the behaviour $b_i$ is a successor term $t'_i$, we simply construe the branch $(\sigma(t'_i, \tilde{t}))$ as a non-terminal transition, via the map $\text{dwc}^{(b)}_X$ which corresponds to the rule (CTXL).

Given a monadic specification, the resulting abstract EIC specification induces (by structural recursion) an operational model $\text{om} : T0 \to MBT0$, an $MB$-coalgebra for the closed terms of the language. Depending on the choice of monad $M = Nc, Te, Ne, Tce, Nce$, the monadic specification corresponds to one of the abstract EIC formats introduced in this section, and the resulting operational model $\text{om}$ is equivalent to a CTS or a syntactic or semantic C(E)TS.
4.3 Semantic Domains and Behavioural Equivalence

All the transition systems introduced in the previous section – the semantic ETS, CTS, and syntactic or semantic CTS’s – were $MB$-coalgebras for various monads $M$, and $BX = V + X$. In the setting of $Cpo_{\perp}$ – where we take $V = \text{Vals}_\perp$ for some collection of $s$-sorted values $\text{Vals}$ – and under the assumption that the monad $M$ is $Cpo_{\perp}$-enriched, by Corollary 3.3.3 $B$ has a lifting $\beta$ to $\text{Kl}(M)$, and the final $B$-coalgebra is given by the initial $B$-algebra, which we have called $D$; it is given by $N \cdot \text{Vals}_\perp$ in $Cpo_{\perp}$. The coalgebra morphisms $\beta_{\text{om}}$ into $D$ have underlying codomain $M D$, and we will take this to be our semantic domain, where the monad $M$ is suitably chosen depending on whether we wish to study the semantics of CTS’s, or syntactic or semantic (c)ETS’s.

In this section, we consider the instantiations of this semantic domain for each class of transition system in $Cpo_{\perp}$, and show that the relevant monads $M$ have the required $Cpo_{\perp}$-enrichment for it to be a final Kleisli coalgebra. We then define behavioural equivalences on programs $p, q$ in terms of the final-coalgebra morphisms $\beta_{\text{om}}$, and also (for semantic (c)ETS’s) the quotienting map $\text{quot}$, to avoid distinguishing semantically equivalent effect trees (such as $x$ or $x$ and $x$).

4.3.1 Semantics and Behavioural Equivalence for CTS’s

We begin by considering CTS’s, where we take $MX = N_{c}X = (X \otimes C1)^{C1}$; this gives a semantic domain of $((N \cdot \text{Vals}_\perp) \otimes C1)^{C1}$. By Corollary 3.3.3, to ensure that this is indeed $(N_{c}$ applied to) the final $B$-coalgebra in the Kleisli category $\text{Kl}(N_{c})$, we need to check that the monad $N_{c}X = (X \otimes C1)^{C1}$ is $Cpo_{\perp}$-enriched, i.e. that it is a locally continuous and strict functor. This follows as both the smash product $\otimes$ and strict function-space $(-)^{C1}$ are locally continuous and strict functors.

The resulting characterisation of program behaviour is as follows. Along the lines of Example 3.3.12, given a CTS $(X, \gamma : X \rightarrow BX)$, the least-fixpoint construction of the final $\overline{B}$-coalgebra morphism $\overline{\beta}_{\gamma}$ shows that each state $x$ in $X$ is assigned a function in $((N \cdot \text{Vals}_\perp) \otimes C1)^{C1}$ which maps each comodel-state $c$ to a tuple $((n, v), c')$ describing the behaviour of state $x$ when evaluated with initial comodel-state $c'$: namely, the number of steps-to-termination $n$, the return value $v$, and the final comodel-state $c'$; or the bottom value $\perp$ if that execution diverges.

Example 4.3.1. Consider the CTS given by an operational model $\text{om} : T_{0} \rightarrow N_{c}BT_{0}$ for While (incorporating auxiliaries like $+_n$), with respect to the canonical comodel given by $S = \mathbb{N}^{L}$. The final $\overline{B}$-coalgebra morphism $\overline{\beta}_{\text{om}} : T_{0} \rightarrow ((N \cdot \text{Vals}_\perp) \otimes C1)^{C1}$ would assign to the program
4.3. Semantic Domains and Behavioural Equivalence

while \(x \leq 10\) do \{ \(x = x + 1;\} \} the following function.

\[
\lambda c.\left\{ \begin{array}{ll}
(5, \_), & \text{if } c(x) > 10 \\
(82 - 7c(x), \_), & \text{if } c(x) \leq 10 \\
\end{array} \right.
\]

The \(\mathcal{B}\)-coalgebra morphism \(\beta_{\text{om}}\) induces a notion of behavioural equivalence \(\equiv_c\) on the states of a cts, as follows:

**Definition 4.3.2.** Two states \(p, q\) of a cts satisfy \(p \equiv_c q\) if they are identified by the final \(\mathcal{B}\)-coalgebra morphism \(\beta\) into the final \(\mathcal{B}\)-coalgebra \(\overline{D}\) in \(\text{Kl}(N_c)\).

One may check by the least-fixpoint construction of \(\overline{\beta}\) that \(p \equiv_c q\) if and only if: for every initial comodel-state \(s, \langle p, s \rangle\) and \(\langle q, s \rangle\) both: (a) terminate with the same final comodel-state \(s'\) and terminal value \(v\) in the same number of steps \(n\); or (b) do not terminate.

**Example 4.3.3.** Consider the following While programs:

\[
p_1 := (x=0; x=1), \quad p_2 := (x=1; x=1), \quad p_3 := (x=1)
\]

In the operational model for While considered as a cts, we have \(p_1 \equiv_c p_2 \not\equiv_c p_3\).

4.3.2 Syntactic and Semantic ets’s

In Section 3.3.2, we described the semantic domain \(T_e(\mathbb{N} \cdot \text{Vals}_\bot)\) for syntactic ets’s at length, given by taking \(M\) to be the free effect-syntax monad \(T_e\). The corresponding coalgebra morphisms assign to each ets-state the effect-tree observed during its execution, with leaves given by pairs \((n, y)\) of the number of steps-to-termination and the return value. In the case of semantic ets’s, one instead takes the monad to be the free-model monad \(N_e\), giving a semantic domain \(N_e(\mathbb{N} \cdot \text{Vals}_\bot)\) which differs in that each program is instead assigned the semantic equivalence class of this effect-tree.

To verify that these semantic domains form a final \(\mathcal{B}\)-coalgebra, we need \(\text{Cpo}_{\bot!}\)-enrichedness of the syntactic effect monad \(T_e\), and the monad \(N_e\) whose identical components \(N_{e_0}\) are induced by a Lawvere theory. \(\text{Cpo}_{\bot!}\)-enrichedness of \(T_e\) was discussed in Section 3.3.2 and proved concretely in Proposition 3.3.11. By contrast, it is less straightforward to verify enrichedness of the monad \(N_{e_0}\). One method is to exploit the connection between ordinary and \((\text{Cpo}_{\bot!})\text{-enriched}\) [HPP06] or discrete [HP06] Lawvere theories (Definitions 3 and 11 respectively). The former differ from ordinary Lawvere theories in that they are given by \(\text{Cpo}_{\bot!}\)-enriched (and suitably structure-preserving) functors \(\mathcal{S}^{op} \to \mathcal{L}\), where \(\mathcal{S}\) is a skeleton of the subcategory of \(\text{Cpo}_{\bot!}\) given by the countably presentable objects, rather than a skeleton of \(\text{Set}\) (Definition 2.3.1); models are
similarly required to be $\mathcal{Cpo}_{\perp}$-enriched. The difficulty is that countably presentable objects are hard to describe in $\mathcal{Cpo}_{\perp}$; to avoid the need for such considerations, discrete Lawvere theories were introduced. Here, one reverts to a skeleton $\aleph_1$ of the category of countable sets; the main difference from an ordinary Lawvere theory is that the inclusion functors $\aleph_1 \to \mathcal{L}$, and models of the theory $\mathcal{L} \to \mathcal{C}$ in category $\mathcal{C}$, are required to be $\mathcal{Cpo}_{\perp}$-enriched functors.

To show $N_{e_0}$ is a $\mathcal{Cpo}_{\perp}$-monad, one may then consider the enriched [HPP06] or discrete [HP06] $\mathcal{Cpo}_{\perp}$-Lawvere theories freely generated by $\mathcal{L}$, and use the results in [HP06] as follows. As argued above, $\mathcal{Cpo}_{\perp}$ is l.c.p., so by Theorems 14 and 15 of [HP06], for either of the freely generated discrete or enriched theories, the forgetful $\mathcal{Cpo}_{\perp}$-functor $U' : \text{Mod}(\mathcal{L'}, \mathcal{C}) \to \mathcal{C}$ has a $\mathcal{Cpo}_{\perp}$-enriched left adjoint which induces a $\mathcal{Cpo}_{\perp}$-monad $N'_{e_0}$. As argued in [HPP06] (after Theorem 2), the unenriched monad corresponding to $N'_{e_0}$ coincides with the monad $N_{e_0} = U'L'F\mathcal{L}$. Hence, we may consider $N_{e_0}$ to be a $\mathcal{Cpo}_{\perp}$-monad, and Corollary 3.3.3 guarantees that $\overline{D}$ is a final $\overline{B}$-coalgebra as required.

**Remark 4.3.4.** It may seem strange that $\mathcal{Cpo}_{\perp}$-enrichedness of $N_{e_0}$ does not require a constraint on the Lawvere theory $\mathcal{L}$, in contrast to the syntactic effect-monad $T_e$, which essentially required that there be no nullary effects in the signature $\text{Eff}$. This is due to a trivial interpretation of nullary effects $e : 0 \to 1$ by models $G$ in $\mathcal{Cpo}_{\perp}$, as the 0-fold product in $\mathcal{Cpo}_{\perp}$ is 0; hence all constants $e$ are uniformly mapped by each model $G$ to the unique arrow $0 \to G1$.

**Example 4.3.5.** In Example 4.1.1, we stated that the free-model monad $N_{e_0}$ for the theory of global store coincided with the multi-sorted side-effect monad $MX = (S \cdot X)^S$, which in turn coincided with the monad $N_eX = (X \otimes C1)^C1$ on $C^S$ induced by the canonical comodel of Example 4.1.9. Thus, for languages with global store, the semantic domain $(S \cdot (N \cdot \text{Vals}_{\perp}))^S$ for semantic $\text{ets}$’s is essentially identical to that for $\text{cts}$’s with respect to the canonical comodel, viz. $((N \cdot \text{Vals}_{\perp}) \otimes C1)^C1$.

**Example 4.3.6.** For languages with non-determinism, quotienting the syntactic effect-trees $T_e$ (containing only binary or’s) gives rise to the free convex powerdomain monad: $N_e = \mathcal{P}_c$. As the initial $B$-algebra $N \cdot \text{Vals}_{\perp}$ is a flat cppo, the resulting semantic domain $\mathcal{P}_c(N \cdot \text{Vals}_{\perp})$ consists of non-empty sets of pairs $(n, v)$ and/or the element $\perp$. Consider an effectful extension of $\text{SWhile}$ with non-determinism, and an operational model $\text{om} : T0 \to T_eBT0$ given by a syntactic $\text{ets}$. The program $\text{while}((0 \text{ or } 1) == 1) \text{ do } \{\text{skip}\}$ would be assigned by the final-coalgebra morphism $\overline{\beta}_{\text{om}}$ to the following effect-tree, in $T_e(N \cdot \text{Vals}_{\perp})$:

$$5, * \text{ or } 10, * \text{ or } 15, * \text{ or } \cdots$$

By contrast, in the corresponding semantic $\text{ets}$, it would be assigned the following element of $\mathcal{P}_c(N \cdot \text{Vals}_{\perp})$. Note that it contains the bottom element $\perp$, explicitly indicating that the
4.3. Semantic Domains and Behavioural Equivalence

A program may never terminate.

\{⊥, (5, *), (10, *), (15, *), \ldots \}

In the same way as before, \( \overline{\beta}_{om} \) induces a behavioural equivalence \( \cong^T_e \) on the states of a syntactic ETS:

**Definition 4.3.7.** Two states \( p, q \) of a syntactic ETS \( \gamma : X \to T_e BX \) satisfy \( p \cong^T_e q \) if they are identified by the final \( \overline{B} \)-coalgebra morphism \( \overline{\beta}_\gamma \) into the final \( \overline{B} \)-coalgebra \( \overline{D} \) in \( \text{Kl}(T_e) \).

One finds that \( p \cong^T_e q \) if and only if both executions produce the same syntactic effect-tree \( \delta \), whose corresponding computation branches either terminate with the same return value in the same number of steps, or both diverge.

**Example 4.3.8.** Consider the extension of \textbf{SWhile} with a single binary \texttt{or} effect. In the operational model for \texttt{sEWhile} – a syntactic ETS– the programs \( p_1 : 1 + (0 \texttt{ or } 1) \) and \( p_2 : (0 \texttt{ or } 1) + 1 \) are both mapped by \( \overline{\beta}_{om} \) to the syntactic effect-tree \((3, 0)\texttt{ or } (3, 1)\) in \( T_e (N \cdot \text{Vals}_\bot) \), hence \( p_1 \cong^T_e p_2 \). By contrast, the program \( p_3 : 1 + (1 \texttt{ or } 0) \) is mapped to a different tree, viz. \((3, 1)\texttt{ or } (3, 0)\), and hence \( p_1 \not\cong^T_e p_3 \).

This illustrates that, in practice, one would wish to relax the restriction that equivalent programs must produce exactly the same syntactic effect-tree; one is generally more interested in semantic equivalence classes of effect trees, as described by models of the Lawvere theory. To remedy this problem, we need to quotient the syntactic effect-trees in the semantic domain \( T_e \overline{D} \) by applying the quotienting map \( \text{quot}_{\overline{T}} : T_e \overline{D} \to N_e \overline{D} \) defined in Section 4.1.2, arriving at a more satisfactory notion of behavioural equivalence for syntactic ETS’s.

**Definition 4.3.9.** Two states \( p, q \) of a syntactic ETS \( \gamma : X \to N_e BX \) satisfy \( p \cong^{T \to N}_e q \) if they are identified by the composition \( X \xrightarrow{\overline{\beta}} T_e \overline{D} \xrightarrow{\text{quot}_{\overline{T}}} N_e \overline{D} \).

**Example 4.3.10.** The above programs \( p_1 : 1 + (0 \texttt{ or } 1) \) and \( p_3 : 1 + (1 \texttt{ or } 0) \) are mapped by \( \overline{\beta}_{om} \) to \((3, 0)\texttt{ or } (3, 1)\) and \((3, 1)\texttt{ or } (3, 0)\) respectively; these in turn are mapped by \( \text{quot}_{\overline{T}} \) to the same set \( \{(3, 0), (3, 1)\} \) in the convex powerdomain over \( \overline{D} \), hence \( p_1 \cong^{T \to N}_e p_3 \).

Alternatively, to ensure semantically equivalent effect-trees are identified, one could first convert a syntactic ETS directly into a **semantic ETS** as described in Section 4.1.2, and then define a behavioural equivalence for semantic ETS’s. We may readily adapt our definitions of the behavioural equivalences \( \cong_c \) and \( \cong^T_e \), to obtain an equivalence \( \cong^N_e \) for semantic ETS’s defined in terms of the final coalgebra morphisms \( \overline{\beta}_\gamma \).

**Definition 4.3.11.** Two states \( p, q \) of a semantic ETS \( \gamma : X \to N_e BX \) satisfy \( p \cong^N_e q \) if they are identified by the final \( \overline{B} \)-coalgebra morphism \( \overline{\beta}_\gamma \) into the final \( \overline{B} \)-coalgebra \( \overline{D} \) in \( \text{Kl}(N_e) \).
As a sanity check, we will show that the behavioural equivalence \( \simeq_{\text{e}}^{T \rightarrow N} \) on a syntactic ETS coincides with the equivalence \( \simeq_{\text{e}}^{N} \) on its translation into a semantic ETS, at the end of Section 4.4.2.

### 4.3.3 Syntactic and Semantic cets’s

We finally consider the case of the cets, where the Lawvere theory \( \mathcal{L} = \mathcal{L}_1 \otimes \mathcal{L}_2 \) is a tensor, \( C \) is a comodel for \( \mathcal{L}_1 \), and the monad \( M \) is taken to be either \( T_{ce}X = (T_{e}^{(2)}(X \otimes C1))^{C1} \) or \( N_{ce}X = (N_{e}^{(2)}(X \otimes C1))^{C1} \) for syntactic and semantic cets’s respectively – where the monads \( T_{e}^{(2)} \), \( N_{e}^{(2)} \) are with respect to the theory \( \mathcal{L}_2 \).

In the case of the syntactic cets, the semantic domain \( T_{ce}\overline{\mathcal{D}} = (T_{e}^{(2)}((N \cdot \text{Val}_1) \otimes C1))^{C1} \) consists of functions which take an initial comodel-state as input, and return a syntactic effect-tree \( \delta(\tilde{x}) \) of effects observed during program execution. Its leaves \( x_i \) describe the computation branches as follows: they take value \( \perp \) whenever that branch diverges; otherwise they are tuples \( (n,\overline{v},c') \) describing the number of steps-to-termination \( n \) at that computation branch, the return value \( \overline{v} \), and the final comodel-state \( c' \). For semantic cets’s, the semantic domain \( T_{ce}\overline{\mathcal{D}} = (N_{e}^{(2)}((N \cdot \text{Val}_1) \otimes C1))^{C1} \) differs only in that the functions essentially return the equivalence class \([\delta(\tilde{x})]\) of an effect-tree.

It remains to show that the monads \( T_{ce} \) and \( N_{ce} \) are \( \text{Cpo}_{\perp\downarrow} \)-enriched; but they are defined by composing functors which we have already shown to be \( \text{Cpo}_{\perp\downarrow} \)-enriched – viz. \( (-)^{C1}, (-) \otimes C1 \), \( T_{e}^{(2)} \) and \( N_{e}^{(2)} \) (for theory \( \mathcal{L}_2 \)). Hence, both monads are \( \text{Cpo}_{\perp\downarrow} \)-enriched, ensuring existence of a semantic domain for both syntactic and semantic cets’s, given by the respective final Kleisli coalgebras.

**Example 4.3.12.** Consider the language \textbf{NDWhile} with an operational model \( \text{om} : T0 \rightarrow T_{ce}BT0 = (T_{e}^{(2)}(BT0 \otimes C1))^{C1} \) given by a syntactic cets, with respect to the canonical comodel for global store in \( \text{Cpo}_{\downarrow} \). The program \textbf{while} ((0 \textbf{ or } 1) == 1) \textbf{do} \{x = 42;\} would then be assigned by \( \overline{\beta}_{\text{om}} \) to the following function:

\[
\lambda c. (\,5,\downarrow,c) \textbf{ or } (10,\downarrow,c[x \rightarrow 42]) \textbf{ or } (15,\downarrow,c[x \rightarrow 42]) \textbf{ or } \ldots
\]

The corresponding semantic cets \( \text{om}' : T0 \rightarrow (N_{e}^{(2)}(BT0 \otimes C1))^{C1} \) is obtained by postcomposing the operational model \( \text{om} \) with \( (\text{quot}_{BT0\otimes C1})^{C1} : (T_{e}^{(2)}(BT0 \otimes C1))^{C1} \rightarrow (N_{e}^{(2)}(BT0 \otimes C1))^{C1} \). In the category \( \text{Kl}(N_{ce}) \), one then has a final \( \overline{B} \)-coalgebra morphism from this semantic cets into the semantic domain \( N_{ce}\overline{\mathcal{D}} = (N_{e}^{(2)}((N \cdot \text{Val}_1) \otimes C1))^{C1} \), where \( N_{e}^{(2)} = \mathcal{P}_{e} \) is the convex powerdomain monad. To interpret this semantic domain more concretely, note that the canonical comodel \( C \) in \( \text{Cpo}_{\downarrow} \) has carrier \( C1 = (C1)_{\downarrow} = (N^{L})_{\downarrow} \), which is a flat cppo, and hence so is the componentwise smash product \((N \cdot \text{Val}_1) \otimes C1 \). Thus, we may characterise the convex
powerdomain \( N^e_2((N \cdot \text{Vals}_\bot) \otimes C1) \) as we did in Example 4.3.6: it consists of sets containing \( \bot \) and/or tuples \((n, \gamma, c)\) where \( n \) is a number of steps-to-termination, \( \gamma \) a return-value, and \( c \) a comodel-state in \( \mathbb{N}^L \). Hence, the semantic domain consists of functions characterising, for each initial comodel-state, the set of possible execution paths – which may include divergence, and/or terminating paths. To illustrate, the above program (in the semantic \( \text{cets om}' \)) is assigned the following function in the semantic domain \( N_{ce}D \):

\[
\lambda c.\{ \bot, (5, \sharp, c), (10, \sharp, c[x \mapsto 42]), (15, \sharp, c[x \mapsto 42]), \ldots \}
\]

Along exactly the same lines as \( \text{ets}'s \), we may define two notions of behavioural equivalence for syntactic \( \text{cets}'s \). The first, viz. \( \equiv^{T \rightarrow N}_{ce} \), is defined solely in terms of the final-coalgebra morphisms \( \overline{\beta} \) into the semantic domain \( (T^e_2(N \cdot \text{Vals}_\bot) \otimes C1)^C1 \), and it identifies two programs \( p, q \) if, for every initial comodel-state \( s \), the configurations \( \langle p, c \rangle \) and \( \langle q, c \rangle \) produce the same syntactic effect-tree \( \delta \), whose corresponding computation branches either: (1) terminate in the same number of steps, returning the same terminal value and the same final comodel state; or (2) both diverge. The second equivalence, viz. \( \equiv^{T \rightarrow N}_{ce} \), further quotients these effect-trees using the monad morphism \( \text{quot}_X : T^e_2X \rightarrow N^e_2X \) (with respect to the Lawvere theory \( \mathcal{L}_2 \)), and it identifies programs which produce the same equivalence class of effect-trees.

Finally, we also define an equivalence \( \equiv^N_{ce} \) for semantic \( \text{cets} \), and we will show that the behavioural equivalence \( \equiv^{T \rightarrow N}_{ce} \) for a syntactic \( \text{cets} \) coincides with the equivalence \( \equiv^N_{ce} \) on its translation into a semantic \( \text{cets} \) (Section 4.1.6), at the end of Section 4.4.2.

**Definition 4.3.13.** Two states \( p, q \) of a syntactic \( \text{cets} \) \( \gamma : X \rightarrow TceBX = (T_e(BX \otimes C1))^C1 \) satisfy \( p \equiv^{T}_{ce} q \) if they are identified by the final \( \overline{B} \)-coalgebra morphism \( \overline{\beta}_\gamma \) into the final \( \overline{B} \)-coalgebra \( D \) in \( \text{Kl}(T_{ce}) \). They satisfy \( p \equiv^{T \rightarrow N}_{ce} q \) if they are identified by the following composition:

\[
X \xrightarrow{\overline{\beta}_\gamma} (T^e_2(N \cdot \text{Vals}_\bot) \otimes C1)^C1 \xrightarrow{(\text{quot}_X \otimes C1)^C1} (N^e_2(N \cdot \text{Vals}_\bot) \otimes C1)^C1.
\]

Two states \( p, q \) of a semantic \( \text{cets} \) \( \gamma : X \rightarrow NceBX = (N_e(BX \otimes C1))^C1 \) satisfy \( p \equiv^{N}_{ce} q \) if they are identified by the final \( \overline{B} \)-coalgebra morphism \( \overline{\beta}_\gamma \) into the final \( \overline{B} \)-coalgebra \( D \) in \( \text{Kl}(N_{ce}) \).

**Example 4.3.14.** Given a syntactic \( \text{cets om} : T0 \rightarrow (T^e_2(BT0 \otimes C1))^C1 \) representing an operational model for \( \text{NDWhile} \), let \( x \) be a store-variable. The commands

\[
p_1 : x = 1 \quad \text{and} \quad p_2 : (x = 1) \text{ or } (x = 1)
\]

are mapped by \( \overline{\beta}_{\text{om}} \) to the respective functions below, of type \( (T^e_2(BT0 \otimes C1))^C1 \).

\[
\lambda c.((1, \sharp, c[x \mapsto 1])) \quad \text{and} \quad \lambda c.((1, \sharp, c[x \mapsto 1]) \text{ or } (1, \sharp, c[x \mapsto 1]))
\]

For all comodel-states, the functions produce different syntactic effect-trees, so \( p_1 \not\equiv^{T}_{ce} p_2 \). By
contrast, the behavioural equivalence $\simeq_{ce}^{T\rightarrow N}$ applies $\text{quot}_{D \otimes C_1} : T_c^{(2)}(D \otimes C_1) \rightarrow N_c^{(2)}(D \otimes C_1)$ to assign each effect-tree to an element of the convex powerdomain $P_c((\mathbb{N} \cdot \text{Vals}_1) \otimes C_1)$ (described in Example 4.3.12) – this element is the singleton set $\{(1, \_ , c[x \mapsto 1])\}$. Thus, both $p_1$ and $p_2$ are mapped to the following function in $(N_c^{(2)}(D \otimes C_1))^{C_1}$, and so $p_1 \simeq_{ce}^{T\rightarrow N} p_2$.

\[ \lambda c. ( \{(1, \_ , c[x \mapsto 1])\} ) \]

### 4.4 Adequacy and Compositionality for Stateful and Effectful Languages

Having introduced a categorical rule format to specify operational models, in the previous section we considered the semantic domains for each class of transition system we have introduced, given by $M\overline{D}$ for various choices of monad $M$, and defined various notions of behavioural equivalence. We now define denotational semantics for each class of transition system, and prove adequacy and compositionality of this denotational semantics for the behavioural equivalences defined in the previous section.

#### 4.4.1 Denotational Models

To make the semantic domain $M\overline{D}$ into a denotational model, we need to provide an interpretation $[\sigma]$ of each syntax constructor $\sigma$ on denotations; categorically, this amounts to a $\Sigma$-algebra structure on $M\overline{D}$. We showed how to do this in Section 3.4.1, in the context of syntactic ETS’s where $M = T_c$; and the construction readily generalises to other monads $M$. We briefly review the construction, which begins by giving a $MB$-coalgebra structure $\tilde{\zeta}$ to the semantic domain $M\overline{D}$; and this is achieved through the composition below.

\[ \tilde{\zeta} : M\overline{D} \xrightarrow{MA^{-1}} MB\overline{D} \xrightarrow{MB\eta\overline{B}} MBM\overline{D} \]

Given an abstract OS specification $\epsilon$, structural recursion then induces an operational model $\overline{\text{omd}} : TM\overline{D} \rightarrow MB(TM\overline{D})$ for syntax terms over denotations, which may also be seen as a $\overline{B}$-coalgebra in the Kleisli category $\text{Kl}(M)$. Hence, there is a final $\overline{B}$-coalgebra morphism $\overline{\beta}_{\overline{\text{omd}}} : TM\overline{D} \rightarrow \overline{D}$ in the Kleisli category, of underlying type $TM\overline{D} \rightarrow M\overline{D}$. Thus, each syntax term over denotations is mapped back into a denotation, providing an interpretation of syntax constructors on denotations as required; as before, we use the following composition to restrict the interpretation to a single layer of syntax, giving a $\Sigma$-algebra structure $\overline{\text{dm}}$ to the semantic
domain $M\overline{D}$ which provides the required interpretations $[\sigma]$ of syntax constructors.

$$
\overline{dm} : \Sigma M\overline{D} \xrightarrow{\Sigma q_M^{\overline{p}}} \Sigma TM\overline{D} \xrightarrow{\psi_M^{\overline{p}}} TM\overline{D} \xrightarrow{\beta_{omd}} M\overline{D}
$$

Example 4.4.1. For cts’s, recall the semantic domain consists of functions $\phi \in N_c\overline{D} = ((N\cdot \text{Vals}_{\perp}) \otimes \text{C1})^C_1$, mapping each initial comodel-state to a characterisation $\overline{D}$ of an execution trace, and a final comodel-state. In a multi-sorted setting, its $s$-sorted component is given by $(N_c\overline{D})_s = N_{\text{om}}(\overline{D}_s) = (\overline{D}_s \otimes \text{C1})^C_1$, where $\overline{D}_s = N\cdot \text{Vals}_s$. Given an abstract EIC specification for the language $\text{While}$, the above construction gives interpretations of the syntax constructors of $\text{While}$ on this semantic domain. For instance, for sequential composition – given by the fragment $(X_N, X_E, X_P) \mapsto X_P \otimes X_P$ of the context-term syntax functor $H_2$ within $\Sigma$ – the map $\overline{dm}$ describes a binary operation $[\cdot] : (N_c\overline{D})_P \otimes (N_c\overline{D})_P \rightarrow (N_c\overline{D})_P$, which essentially ‘chains together’ two given (command-type) denotations $\phi_1, \phi_2 \in (\overline{D}_P \otimes \text{C1})^C_1$. To illustrate, consider the $\text{While}$ programs $p_1 : (x = 1; \text{skip}; \text{skip})$ and $p_2 : (y = 2; \text{skip})$. They are mapped by $\overline{dm}$ to the functions

$$
\phi_1 : \lambda c.(3, x, c[x \mapsto 1]) \quad \phi_1 : \lambda c.(2, x, c[y \mapsto 2]).
$$

The interpretation of sequential composition $[\cdot]$ on the denotations, $\phi_1[\cdot]\phi_2$, is the function which, given an initial comodel-state $c$, supplies it as input to $\phi_1$, giving a number $n_1$ of steps-to-termination and a new comodel-state $c'$. The latter is used as the input of $\phi_2$, giving the final state $c''$ and another number $n_2$ of steps-to-termination, which is added to $n_1$ to give the overall steps-to-termination. (The return value $x$ plays little part in this process.) For the above examples, the resulting function is shown below.

$$
\phi_1[\cdot]\phi_2 : \lambda c.(5, x, c[x \mapsto 1, y \mapsto 2])
$$

Example 4.4.2. Denotational models for syntactic ETS’s were considered in Example 4.3.6. Here, we consider a semantic ETS: the operational model for $\text{SWWhile}$, extended with nondeterminism. Recall that the semantic domain $N_c\overline{D} = \mathcal{P}_c(N\cdot \text{Vals}_{\perp})$ consists of sets containing $\perp$ and/or pairs $(n, v)$. Suppose two numeric expressions $p_1, p_2$ have denotations $S_1, S_2$; then the interpretation of addition $+\,\text{on the denotations}$, $S_1 + S_2$, is the set

$$
\{(n_1 + n_2, v_1 + v_2) : (n_1, v_1) \in S_1 \text{ and } (n_2, v_2) \in S_2\} \cup \{(\perp) \cap (S_1 \cup S_2)\}
$$

expressing the fact that if $p_1$ may take $n_1$ steps to produce value $v_1$, and similarly if $p_2$ may take $n_2$ steps to produce $v_2$, then their sum $p_1 + p_2$ may take $n_1 + n_2$ steps to produce $v_1 + v_2$; furthermore, if either $p_1$ or $p_2$ is capable of divergence, then so is $p_1 + p_2$.

Example 4.4.3. We now consider a syntactic cets: the operational model for $\text{NDWhile}$. The semantic domain consists of functions $\phi : (\mathcal{P}_c((N\cdot \text{Vals}_{\perp}) \otimes \text{C1}))^C_1$, assigning initial comodel-states $c$ to binary or-trees, whose leaves are either divergence $\perp$ or tuples $(n, v, c')$ describing
each execution path. The interpretation \([\text{or}]\) of the syntactic \textbf{or} operator on functions \(\phi_1, \phi_2\),
given an initial comodel-state \(c\), essentially combines the effect-trees \(\phi_1(c), \phi_2(c)\) by applying
the \textbf{or} operator, and increments the steps-to-termination \(n\) at each branch. For instance, the
denotation \(\phi_n\) of the program \(x = n\) is \(\lambda c. \eta^L_x(1, \_ \_c[x \mapsto n])\), where the unit \(\eta^L_x\) indicates
that there is no non-determinism, and the denotation of \textbf{skip} is \(\lambda c. (\eta^L_x(1, \_ \_c) – \text{ hence}
\phi_1 \ [\text{or}] \ \textbf{skip} = \lambda c. (2, \_ \_c[x \mapsto 1]) \text{ or } (2, \_ \_c)\).

We may also convert the operational model for \textbf{NDWhile} into a \textit{semantic} \textbf{cets}. The theory
\(L_2\) of non-determinism gives rise to the convex powerdomain monad \(N \(= \mathcal{P}\), and hence the
semantic domain for semantic \textbf{cets}’s, \((\mathcal{P}((\mathbb{N} \cdot \text{Vals}_\perp) \otimes \text{C}^1))\)\(^C\) describes functions mapping
comodel-states to \textit{sets} of tuples \((n, \_ \_c)\) and/or \(\perp\). The interpretation \([\text{or}]\) of the choice oper-
ator now involves taking unions of these sets (rather than applying a syntactic \textbf{or}-effect), and
incrementing the values of \(n\). For instance, the above example becomes:

\[
\phi_1 \ [\text{or}] \ \phi_2 = \lambda c. (\{2, \_ \_c[x \mapsto 1]\}, (1, \_ \_c)\).
\]

\[4.4.2 \ \text{A Monadic Approach to Adequacy and Compositionality}\]

As discussed in Section 3.4.1, having defined a denotational model \(\overline{\text{om}} : \Sigma M \overline{D} \rightarrow M \overline{D}\), the
initial \(\Sigma\)-algebra \(T_0\) then gives rise to a \(\Sigma\)-algebra morphism \([\text{-}]\), allowing us to inductively assign
denotations \([\sigma((t_i)_{i \in I})]\) to terms \(\sigma((t_i)_{i \in I})\) in a modular fashion, by applying the interpretation
\([\sigma]\) of the outer-most constructor \(\sigma\) to the denotations \([t_i]\) of its arguments. The assign-
ment of denotations is necessarily \textit{compositional}, in that \([\sigma((t_i)_{i \in I})]\) = \([\sigma]\)(([t_i])_{i \in I})\).

As before, our goal is to prove \textit{adequacy}: that denotational implies behavioural equivalence. For
each class of transition system, two programs are denotationally equivalent whenever they are
identified by the map \([\text{-}]\). We have different notions of behavioural equivalence: we introduced
the relation \(\cong\) for the CTS, \(\cong^T\) and \(\cong^N\) for the syntactic and semantic CTS respectively, and
\(\cong^T_{ce}\) and \(\cong^N_{ce}\) for the syntactic and semantic \textbf{cets} respectively. These five equivalences were all
characterised by the maps \(\overline{\beta}^{M}_{\text{om}}\) into the respective final Kleisli coalgebras in \(\text{Kl}(M)\) for various
monads \(M – \text{which, in this section, we sometimes distinguish by the decoration }^M\) as shown.
Hence, to prove adequacy for these cases, it is sufficient to prove that \(\overline{\beta}^{M}_{\text{om}} = [\text{-}]\); and as before,
this would be implied if we could show that \(\overline{\beta}^{M}_{\text{om}}\) was a \(\Sigma\)-algebra morphism \(T_0 \rightarrow M \overline{D}\) like
\([\text{-}]\), by initiality of \(T_0\). This is implied by the main theorem of this chapter, which we prove
in Section 4.4.3.

\textbf{Theorem 4.4.4.} Suppose we are given a set of sorts \(S\) and a syntax signature \(\text{Sig}_c\) for context-
terms. Let \(C\) be a symmetric monoidal closed category, \(M\) a strong monad on \(C^S\) given compo-
nentwise by a strong monad \(M_0\) on \(C\), \(B\) an endofunctor \(BX = V + X\) with lifting \(\overline{B}\) to \(\text{Kl}(M)\),
and $\Sigma$ an endofunctor in $R$-C form with respect to $R, \Sigma$; and suppose the final $B$-coalgebra is $(D, \eta^B \circ \alpha^{-1})$ where $D$ is the initial $B$-algebra on $C$, and $\alpha : BD \rightarrow D$ is its algebra structure. Given an abstract EIC specification $\epsilon$ (Definition 4.2.19) with respect to monad $M$, inducing an operational model $\text{omt} : T0 \rightarrow MBT0$, the underlying arrow $\beta_{\text{omt}} : T0 \rightarrow MD$ of the final $B$-coalgebra morphism into $D$ is a $\Sigma$-algebra morphism.

**Corollary 4.4.5.** The denotational semantics for CTS’s and syntactic or semantic (c)ETS’s is adequate and compositional with respect to the behavioural equivalences $\simeq_{e$, $\simeq_T$, $\simeq_{ce}$, $\simeq_{e$, $\simeq_{ce}$.

We have also introduced two further behavioural equivalences $\simeq_{T \rightarrow N}$, $\simeq_{ce}$ for syntactic (c)ETS’s, which compare the effect-trees produced by programs whilst taking the equational effect-theory into account, resulting in assignments into the denotational models for semantic (c)ETS’s. These equivalences were instead characterised by maps $m_{\text{om}} \circ \beta^M_{\text{om}}$ which post-compose the coalgebra morphisms $\beta^M_{\text{om}}$ with monad morphisms $m : M \Rightarrow N$, where respectively $m_X = \text{quot}_X : T_X \Rightarrow N_X$ and $m_X = (\text{quot}_X \otimes C1)^C1 : (Te^2(X \otimes C1))C1 \rightarrow (Ne^2(X \otimes C1))C1$. To prove adequacy, viz. $[-] = m \circ \beta^M_{\text{om}}$, we will need to prove that the above final-coalgebra semantics is coherent with respect to these monad morphisms. We will do this by proving that

$$m \circ \beta^M_{\text{om}} = \beta^N_{\text{omt}}$$

where $\text{omt}$ (‘operational model translation’) is the semantic (c)ETS given by ‘translating’ the Syntactic Abstract EIC2 or 3 specification for the language, via the monad morphism $m$, into a Semantic Abstract EIC2 or 3 Specification. To illustrate how this translation works, recall that all of these specifications are equivalent to monadic EIC specifications, for various monads $M$; and these are given by natural transformations $r_X : RX \Rightarrow MBTX$ and $e_X : H(V, X) \Rightarrow MBTX$. Hence, if one has such a monadic EIC specification with respect to $M$, and a monad morphism $m : M \Rightarrow N$, one may post-compose these maps with $m_{BTX} : MBTX \rightarrow NBTX$ to obtain a new monadic EIC specification $r'_X : RX \Rightarrow NBTX$ and $e'_X : H(V, X) \Rightarrow NBTX$ with respect to $N$. Thus, if an abstract EIC specification $e_X^M : \Sigma(TX \times MBTX) \rightarrow MBTX$ is induced by the former monadic specification with respect to $M$ (via Definition 4.2.19), then by translating that into a monadic specification with respect to $N$, one obtains a corresponding “translated” abstract EIC specification $e_X^N : \Sigma(TX \times NBTX) \rightarrow NBTX$.

**Proposition 4.4.6.** Let $M$ and $N$ be strong monads with costrengths $\text{cost}^M$, $\text{cost}^N$ – given componentwise by monads $M_0, N_0$ with costrengths $\text{cost}^M_0$, $\text{cost}^N_0$ – both satisfying the assumptions of Theorem 4.4.4. Write $B^M$ and $B^N$ for the liftings of the endofunctor $BX = V + X$ to $\text{Kl}(M)$ and $\text{Kl}(N)$ respectively. Finally, let $m : M \Rightarrow N$ be a strong monad morphism, with components $m^{(0)} : M_0 \Rightarrow N_0$. This means that in addition to the diagrams of Lemma 4.1.5, the
Given an abstract EIC specification \( \epsilon^M \) with respect to monad \( M \), and its translation via \( m \) into a specification \( \epsilon^N \) with respect to monad \( N \), the corresponding final coalgebra maps \( \beta^M \), \( \beta^N \) from the induced operational models \( \text{om} : T_0 \to MBT_0, \text{omt} : T_0 \to NBT_0 \) into \( MD \) and \( ND \) satisfy \( \beta^N \circ \text{omt} = m \circ \beta^M \circ \text{om} \), provided the distributive laws \( \lambda^M, \lambda^N \) lifting \( B \) make the following diagram commute.

\[
\begin{array}{ccc}
BMX & \xrightarrow{Bm_X} & BNX \\
\downarrow^{\lambda^M_X} & & \downarrow^{\lambda^N_X} \\
MBX & \xrightarrow{mBX} & NBX
\end{array}
\]

\[ (\dagger) \]

Proof. Recall that an abstract EIC specification \( \epsilon^M : \Sigma(TX \times MBTX) \to MBTX \) is induced by a monadic EIC specification with respect to \( M \) – viz. a pair of natural transformations \( r_X : RX \to MBTX \) and \( e_X : H(V,X) \to MBTX \); we can post-compose these maps with \( m_{BTX} : MBTX \to NBTX \) to obtain a monadic EIC specification with respect to \( N \), and this induces the translated EIC specification \( \epsilon^N : \Sigma(TX \times NBTX) \to NBTX \).

We will use superscripts \( M, N \) to distinguish the natural transformations \( r^M, \text{dwc}^M, \text{dwc}^M \), etc. of Definitions 4.2.16 and 4.2.19 used to induce the abstract EIC specification \( \epsilon^M \), from the corresponding natural transformations \( r^N, \text{dwc}^N, \text{dwc}^N \) which occur in the definition of the translated specification \( \epsilon^N \). For instance, we have \( r^N_X = m_{BTX} \circ r^M_X : RX \to MBTX \).

\( \beta^N \circ \text{omt} \) is the final \( B^N \)-coalgebra morphism from the operational model \( \text{omt} : T_0 \to NBT_0 \) induced by the specification \( \epsilon^N \), into the final \( B^N \)-coalgebra \( D \), with structure \( \eta^N \circ \alpha^{-1} \). To prove \( \beta^N \circ \text{omt} = m_{T_0} \circ \beta^M \circ \text{om} \), we show that the right-hand side is also a \( B^N \)-coalgebra morphism from \( T_0 \) into \( D \); by finality of \( D \), it must coincide with \( \beta^N \circ \text{omt} \).

Recalling that the semantic map \( \beta^M \circ \text{om} \) is a \( B^M \)-coalgebra morphism from the effectful operational model \( \text{om} : T_0 \to MBT_0 \) to the final \( B^M \)-coalgebra – also \( D \), but with structure \( \eta^M \circ \alpha^{-1} \) – this strategy amounts in the underlying category to proving commutativity of the outer diagram below. We write \( \dagger_M \) and \( \dagger_N \) to distinguish the dagger operations from each Kleisli category; and we have used the fact that \( (\eta^M \circ \alpha^{-1}) \dagger_M = M\alpha^{-1} \) – see Remark 2.2.4 for details – and
similarly for $N$.

The central top square commutes by definition of $\beta_{\text{om}}^M$ as a $\overline{B}^M$-coalgebra morphism, and the right-hand square by naturality of the monad morphism $m$. The bottom square corresponds to the outer paths of the following diagram.

The rightmost square commutes as $m$ is a monad morphism, and the bottom-central square is $N$ applied to the compatibility condition $\dagger$ in the statement of the proposition. The other parts commute by naturality of $m$.

It remains to prove commutativity of the triangle $\dagger$ in the first diagram. Recalling that $\text{omt}$ is given uniquely by structural recursion (Section 2.1.8 and Definition 4.2.19), we prove that $m_{BT0} \circ \text{om}$ also satisfies the same diagram, shown left below. The bottom half is trivial as $0$ is initial. The top half may be reduced as in the right diagram: here, the left square is part of the definition of $\text{om}$. It remains to prove the right square (A).

The syntax functor $\Sigma$ is necessarily in EIC form – $\Sigma X = RX + H_2X$ – so we may prove (A) by considering each component of this coproduct. The reduct case is handled in the following diagram. The arrows along the left-hand edge correspond to $\epsilon_{T0}^M$, and those on the far-right edge correspond to $\epsilon_{T0}^N$. From top to bottom, the parts commute by: properties of products; the definition of the translated redex specification data $r^N$; and naturality of $m : M \Rightarrow N$.
respectively.

\[
R(T0 \times MBT0) \xrightarrow{R(\text{id} \times m_{BT0})} R(T0 \times NBT0)
\]

\[
\xrightarrow{R \sigma_1} \xrightarrow{R \sigma_1}
\]

\[
MBT^20 \xrightarrow{m_{BT^20}} NBT^20
\]

\[
M_{\mu_{BT0}} \xrightarrow{\mu_{BT0}} N_{\mu_0}
\]

As for the case of context terms, it may be broken down as follows. Excluding the square (B) and triangle (C), top-to-bottom the parts commute as follows: (1) properties of products; (2, or ‘B’) proved below; (3, 4) naturality of \( m \); (5) part of the definition of a monad morphism \( m : M \Rightarrow N \).

\[
H_2(T0 \times MBT0) \xrightarrow{H_2(\text{id} \times m_{BT0})} H_2(T0 \times NBT0)
\]

\[
\xrightarrow{H(\pi_2, \pi_1)} \xrightarrow{\text{cost}^{MBT0,T0}_{\text{dist}^{V,T0}_{\text{dwc}^{T0}}}_{\text{\sigma}_{\text{\mu}_{\text{BT0}}}}}
\]

\[
M(H(V,T0) + H_2 T0) \xrightarrow{\text{cost}^{MBT0,T0}_{\text{\mu}_{\text{BT0}}}} N(H(V,T0) + H_2 T0)
\]

\[
\xrightarrow{m_{MBT0}} \xrightarrow{\text{cost}^{MBT0,T0}_{\text{\mu}_{\text{BT0}}}}
\]

\[
\text{dist}^{V,T0}_{\text{dwc}^{T0}} \xrightarrow{\text{\sigma}_{\text{\mu}_{\text{BT0}}}} \text{\mu}_{\text{BT0}}
\]

We break down the \( s \)-component of (B) as follows, where we again write \( \prod_\sigma \) instead of \( \prod_{\sigma:s_0, (s_i)\to s_i} \), and \( Z_\sigma \) instead of \( \prod_i Y_{s_i} \); the top square is a \( \sigma \)-fold coproduct of one of the axioms for a strong monad morphism \( m^{(0)} \), as displayed in the statement of this proposition (for \( m \)).
In the lower square, we show that the top-right path is the same as the bottom-left path:

\[ [N_0 \text{inj}_\sigma]_\sigma \circ \prod_\sigma m^{(0)}_{X_\sigma \otimes Z_\sigma} = [N_0 \text{inj}_\sigma \circ m^{(0)}_{X_\sigma \otimes Z_\sigma}]_\sigma = [m^{(0)}_{X_\sigma \otimes Z_\sigma} \circ M_0 \text{inj}_\sigma]_\sigma = m^{(0)}_{X_\sigma \otimes Z_\sigma} \circ [M_0 \text{inj}_\sigma]_\sigma \]

The first step uses the identity \([g_\sigma]_\sigma \circ \prod_\sigma f = [g_\sigma \circ f]_\sigma\); the second uses naturality of \(m^{(0)} : M_0 \Rightarrow N_0\); and the last step uses the identity \([f \circ g_\sigma]_\sigma = f \circ [g_\sigma]_\sigma\).

It remains to prove commutativity of (C), which we do with one application of \(N\) removed, as shown below. The bottom paths correspond to \(m \circ \text{dwc}^{(v)}_N^{M_{T_0}}\) and \(m \circ \text{dwc}^{(b)}_N^{M_{T_0}}\), and the top paths to \(\text{dwc}^{(v)}_N^{M_{T_0}}\) and \(\text{dwc}^{(b)}_N^{M_{T_0}}\) respectively. By definition, the translated specification data \(e_T^{N_N}\) is given by post-composing \(e_T^{M_{T_0}}\) with \(m_{BT^{20}}\), so that the top-left triangle commutes; the other parts commute as \(m\) is a monad morphism.

\[ \begin{array}{c}
H(V, T_0) \\
\downarrow m_{BT^{20}}^N \\
H(V, T_0) \\
\downarrow m_{BT^{20}}^N \\
H(V, T_0) \\
\downarrow m_{BT^0}^N \\
BT^0 \\
\downarrow m_{BT^0}^N \\
BT^0 \\
\downarrow m_{BT^0}^N \\
BT^0 \\
\downarrow m_{BT^0}^N \\
BT^0 \\
\downarrow m_{BT^0}^N \\
\end{array} \]

We now show that the previous result implies adequacy for syntactic \((C)\text{ETS}'s\) with respect to \(\simeq_{(T^e)^N}, \simeq_{(T^e)^N}\) respectively.

**Corollary 4.4.7.** The conditions of Proposition 4.4.6 are met in the setting of \(\text{Cpo}_{\bot^T}\), where we take \(M = T_0\) or \(T_{ce}\), \(N = N_e\) or \(N_{ce}\), and the respective monad morphisms \(\text{quot} : T_0 \Rightarrow N_e\) and \((\text{quot}_{(-)\otimes C_1})^{T_{ce}} : T_{ce} \Rightarrow N_{ce}\). This implies

\[ [-]^{N_e} = \beta^{N_e}_{\text{omt}} = \text{quot}_{T_0} \circ \beta^{T_{ce}}_{\text{omt}} \quad \text{and} \quad [-]^{N_{ce}} = \beta^{N_{ce}}_{\text{omt}} = (\text{quot}_{T_0 \otimes C_1})^{T_{ce}} \circ \beta^{T_{ce}}_{\text{omt}} \]

which implies adequacy and compositionality of the denotational semantics for syntactic \((C)\text{ETS}'s\) with respect to \(\simeq_{(T^e)^N}, \simeq_{(T^e)^N}\).

These equations also imply that the behavioural equivalences \(\simeq_{(T^e)^N}, \simeq_{(T^e)^N}\) for operational models \(\text{om}\) given by syntactic \((C)\text{ETS}'s\) (characterised by the compositions on the right-hand sides) coincide with the equivalences \(\simeq_{(T^e)^N}, \simeq_{(T^e)^N}\) on their translations \(\text{omt}\) into semantic \((C)\text{ETS}'s\) (characterised by the maps in the middle).

**Proof.** We first recall that all the monads \(M\) we introduced are \(\text{Cpo}_{\bot^T}\)-enriched; hence Corollary
3.3.3 applies, and the constraints on the final $B^M$- and $B^N$-coalgebras are met where $B$ is the initial $B$-algebra, with algebra-structure $\alpha$. The condition (1) is easily verified for the liftings $B$ defined by the distributive law of Example 2.2.10 for each monad $M$ and $N$: it amounts to the left-most diagram below. It may be verified for each component of the top-left coproduct $V + MX$, giving the following two diagrams, which commute by definition of monad morphisms and naturality of $\text{inr}_X : X \rightarrow V + X$ respectively.

Finally, for the requirement that $m$ be a strong monad morphism, note that if $M$ and $N$ are enriched $\mathcal{V}$-monads (for symmetric $\otimes$-monoidal $\mathcal{V}$), then the monad morphism $m$ is strong if and only if it is a $\mathcal{V}$-natural transformation between the monads (see Remark 1.4 of [Koc72]). We use the fact that $\mathcal{V}$-naturality between $\mathcal{V}$-functors $F, G$ is equivalent to ordinary naturality between their underlying ordinary functors $F_0, G_0$ if the functor $V = \mathcal{V}(I, -) : \mathcal{V} \rightarrow \text{Set}$ is faithful, where $I$ is the monoidal unit of $\otimes$ ([Kel05] Section 1.3); and in this case, any monad morphism $m$ between strong monads is itself strong.

This property is straightforward to verify when $\mathcal{V} = \text{Cpo}_{\perp}$, where $I$ is the two-element cppo $1_{\perp}$, and the set $\text{Cpo}_{\perp}(I, A)$ corresponds to functions $1 \mapsto a$ ‘picking out’ the elements $a$ of the cppo $A$ (including $\perp$). To show that $V$ is faithful, if two arrows $f, g : A \rightarrow B$ in $\text{Cpo}_{\perp}$ are distinct, then some element $a$ of the cppo $A$ must have $f(a) \neq g(a)$. In turn, the arrows $\text{Cpo}_{\perp}(I, f), \text{Cpo}_{\perp}(I, g) : \text{Cpo}_{\perp}(I, A) \rightarrow \text{Cpo}_{\perp}(I, B)$ map the function $1 \mapsto a$ to the distinct functions $1 \mapsto f(a)$ and $1 \mapsto g(a)$ respectively, hence $\text{Cpo}_{\perp}(I, f) \neq \text{Cpo}_{\perp}(I, g)$.

With this, we have proved adequacy and compositionality for all the behavioural equivalences for the classes of transition systems we have introduced.

### 4.4.3 Proof of Theorem 4.4.4

Given an abstract EIC specification $\epsilon$ inducing an operational model $\text{om}$, our goal is to prove that the underlying arrow of the $\overline{\beta}$-coalgebra morphism $\overline{\beta}_{\text{om}} : T0 \rightarrow M\overline{D}$ from the operational model into the final $\overline{B}$-coalgebra, $\overline{D}$, of type $T0 \rightarrow M\overline{D}$, is a $\Sigma$-algebra morphism into the $\Sigma$-algebra structure $\overline{\text{dm}}$ on $M\overline{D}$ (Section 4.4.1).

In the previous chapter (Section 3.4.2), we exploited adequacy and compositionality of the semantics induced by the final $MB$-coalgebra $(D, \zeta)$, and reduced the problem to commutativity
of the following square. Once again, $\beta_\zeta : D \to MBD$ is the underlying arrow of the final $B$-
coclassification morphism from the final $MB$-cocom芬兰 $D$ into $\overline{D}$; $\text{omd}$ is the operational model
$T^e(\zeta) : TD \to MBTD$ for fine-grained denotations (induced by the abstract OS specification $e$, by structural recursion over denotations $D$); and $\overline{\text{omd}}$ is the operational model $T^f(\tilde{\zeta}) : TM\overline{D} \to MBTM\overline{D}$ for coarse-grained denotations, this time by structural recursion over
the $MB$-cocom芬兰 structure for coarse-grained denotations, viz. $\tilde{\zeta} : M\overline{D} \xrightarrow{M\alpha^e} MB\overline{D} \xrightarrow{MB\eta^f} MBM\overline{D}$.

$$
\begin{array}{ccc}
TD & \xrightarrow{\beta_\text{omd}} & D \\
\downarrow T\overline{\beta_\zeta} & (+) & \downarrow \overline{\beta_\zeta} \\
TM\overline{D} & \xrightarrow{\overline{\beta_\text{omd}}} & M\overline{D}
\end{array}
$$

In Section 3.4.3, by arguing in terms of cones in the Kleisli category, we showed that this
condition was equivalent to commutativity of the following diagram.

$$
\begin{array}{cccccc}
TD & \xrightarrow{JT\overline{\beta_\zeta}} & MT\overline{D} & \xrightarrow{M\!\beta_{TM\overline{D}}} & M0 \\
\downarrow \text{omd} & & \downarrow T\!\beta_{\text{omd}} & & \downarrow (\text{omd})^f \\
MBTD & \xrightarrow{MB\!\beta_{TM\overline{D}}} & MBT\overline{D} & \xrightarrow{MB\!\beta_{2TM\overline{D}}} & MB0 \\
\downarrow (\text{omd})^f & & \downarrow MB\!\beta_{2TD} & & \downarrow (\text{omd})^f \\
MB^2TD & \xrightarrow{MB^2\!\beta_{TM\overline{D}}} & MB^2T\overline{D} & \xrightarrow{MB^2\!\beta_{2TD}} & MB^20 \\
\vdots & & \vdots & & \vdots
\end{array}
$$

A tempting strategy is to try and connect the left and right columns in a way that ensures
the resulting diagram commutes. The obvious starting point is to check if the square shown
below-left commutes. (The bottom equality is part of the definition of liftings $B$ to $Kl(M)$.)
This is equivalent to the right-hand square $(+)$ in the underlying category, by Remark 2.2.4.

$$
\begin{array}{ccc}
TD & \xrightarrow{JT\overline{\beta_\zeta}} & TMD \\
\downarrow \text{omd} & (+) & \downarrow \text{omd} \\
BTD & \xrightarrow{B\!JT\overline{\beta_\zeta} = JB\overline{\beta_\zeta}} & BTM\overline{D} \\
\downarrow \text{omd} & & \downarrow \text{omd} \\
MBTD & \xrightarrow{MB\!\beta_{TM\overline{D}}} & MBT\overline{D} \\
\downarrow MB\!\beta_{2TD} & & \downarrow MB\!\beta_{2TD} \\
MB^2TD & \xrightarrow{MB^2\!\beta_{TM\overline{D}}} & MB^2T\overline{D}
\end{array}
$$

In general, it does not commute. However, by considering why it does not, through the
restricted mechanics of EIC specifications for redex and context terms, we will arrive at a way of
connecting the columns and produce a commuting diagram, by introducing a more structured
notion of effectful transition behaviour with binary indicators; this allows us to define a suitable
map $\hat{c}$ to connect the columns in a commuting way.
Chapter 4. Semantics for Comodels and Effects

We motivate the definition of \( \hat{\cdot} \) with some examples, in the context of SWhile extended with syntactic effects for global store and non-determinism (i.e. given by a syntactic ets). First, we let \( d, e, f \) be the fine-grained denotations in \( D \), of command sort and with the following transitions:

\[
d \rightarrow \text{wr}_y,6(\#) \quad e \rightarrow \text{wr}_{x,5}(d) \rightarrow \text{wr}_{x,5}(\text{wr}_{y,6}(\#))
\]

\[
f \rightarrow \# \text{ or } f \rightarrow \# \text{ or } (\# \text{ or } f) \rightarrow \ldots
\]

Using the operational specification for sequential composition, we derive transitions for the following terms over fine-grained denotations, in \( TD \) – the first (syntactic or) is an example of a redex, and the second is a context term:

\[
d \text{ or } e \rightarrow d \text{ or } e
\]

\[
(f; d) \rightarrow (d \text{ or } (f; d)) \rightarrow (\text{wr}_{y,6}(\#) \text{ or } (d \text{ or } (f; d))) \rightarrow (\text{wr}_{y,6}(\#) \text{ or } (\text{wr}_{y,6}(\#) \text{ or } (d \text{ or } (f; d)))) \ldots
\]

Now the denotations \( d, e, f \) are mapped by \( \beta_\zeta \) to coarse-grained denotations \( \bar{d}, \bar{e}, \bar{f} \) respectively in \( T_{\bar{c}}D \), with the following transitions, where we refer to the coarse-grained denotation \( s_n \in T_{\bar{c}}D \), for all \( n \in \mathbb{N} \), with effectless transitions \( \bar{s}_1 \rightarrow \# \) and \( \bar{s}_{n+1} \rightarrow \bar{s}_n \). As before, the entire effect-tree is observed in the first execution step.

\[
\bar{d} \rightarrow \text{wr}_{y,6}(\#) \quad \bar{e} \rightarrow \text{wr}_{x,5}(\text{wr}_{y,6}(\bar{s}_1)) \rightarrow \text{wr}_{x,5}(\text{wr}_{y,6}(\#)) \]

\[
\bar{f} \rightarrow \# \text{ or } (\bar{s}_1 \text{ or } (\bar{s}_2 \text{ or } \ldots)) \rightarrow \# \text{ or } (\# \text{ or } (\# \text{ or } \ldots)) \rightarrow \ldots
\]

The same operational specification now induces transitions for the corresponding redex and context-term, in \( TT_{c}D \), as follows:

\[
\bar{d} \text{ or } \bar{e} \rightarrow \bar{d} \text{ or } \bar{e}
\]

\[
(\bar{f}; d) \rightarrow (\bar{d} \text{ or } ((\bar{s}_1; d) \text{ or } ((\bar{s}_2; d) \text{ or } \ldots))) \rightarrow \text{wr}_{y,6}(\#) \text{ or } (\bar{d} \text{ or } ((\bar{s}_1; d) \text{ or } \ldots))) \rightarrow \text{wr}_{y,6}(\#) \text{ or } (\text{wr}_{y,6}(\#) \text{ or } (\bar{d} \text{ or } \ldots)) \rightarrow \ldots
\]

We now show that the diagram (+) commutes for redexes, but not for denotations or context-terms, as illustrated by the examples below. (We have omitted injections \( \text{inl}, \text{inr} \) for clarity, except in the first diagram which is repeated with them included. In the first diagram, we have reduced all the inductively defined arrows to their base-cases: i.e. we omit the unit \( \eta_{T_c} \) of the syntactic monad \( T_c \), writing e.g. \( d \) and \( \bar{d} \) instead of \( \eta_{T_c}(d) \) and \( \eta_{T_{\bar{c}}D}(\bar{d}) \), and similarly \( \zeta \) instead of \( \text{omd} \), \( \tilde{\zeta} \) instead of \( \overline{\text{omd}} \), and \( T_cB\beta_\zeta \) instead of \( T_cBT\beta_\zeta \).)
To make the first diagram commute, we may replace the bottom arrow $T_eB\beta_\zeta$ with the underlying arrow $(B\beta_\zeta)^\dagger$ given by $(B$ applied to) the final $B$-coalgebra morphism $\beta_\zeta : D \rightarrow \overline{D}$, shown below; this replaces the diagram with the following square, and it is easy to show that it commutes by the fact that $\beta_\zeta$ is a $B$-coalgebra morphism from $(D, \zeta)$ to $(M\overline{D}, \tilde{\zeta})$.

$$(B\beta_\zeta)^\dagger : T_eBD \xrightarrow{T_eB\beta_\zeta} T_eBT_e\overline{D} \xrightarrow{T_e\lambda\overline{\pi}} T_e^2B\overline{D} \xrightarrow{\mu\overline{\pi}} T_eB\overline{D}$$

To show why this fixes the first diagram above in more detail, we temporarily make explicit the injections $\text{inl}, \text{inr}$, where we use the fact that, as an element of $T_e\overline{D}$, we may formally identify $d$ with $\text{wr}_{y,6}(\overline{s_1})$. The square then becomes as shown; the maps $\mu_{B\overline{D}} \circ T_e\lambda\overline{\pi}$ may be thought of as ‘pulling out effects’ from the denotation $\overline{d}$, and making them observed rather than ‘hidden’.
inside \(\overline{d}\).

\[
\begin{array}{c}
\zeta \\
\downarrow \\
wr_{x,5}(\text{inr}(d))
\end{array}
\quad
\begin{array}{c}
\overline{\beta}_\zeta \\
\downarrow \\
wr_{x,5}(\text{inr}(\overline{d})) = \mu_{T,B}\circ T,\lambda_{\overline{\beta}}(wr_{y,6}(\overline{\sigma}_1)))
\end{array}
\quad
\begin{array}{c}
\zeta \\
\downarrow \\
\end{array}
\]

We now consider how to make the third diagram commute, for the context term \((f; d)\). It has the active argument \(f\) which, in the fine-grained operational model \(\text{omd}\), non-deterministically has either a terminal transition to \(\ast\) or a non-terminal transition to \(f\), represented by the branches of \(\ast\) or \(f\). The term \((f; d)\) has a corresponding transition to \(d\) – given by the part of the operational specification which handles termination of the active argument (the syntactic rules \((\text{CTXR})\) of Section 4.2.1, or more abstractly, the natural transformation \(e\) in Definition 4.2.16) – and a transition to \((f; d)\), given by putting the successor state \(f\), from the non-terminal transition \(f \rightarrow f\), back into the context \((-); d\) (as represented by rule \((\text{CTXL})\)).

These non-deterministic outcomes are together represented by the transition \((f; d) \rightarrow d\) or \((f; d)\); but we need to handle these outcomes separately in order to convert this transition into the required behaviour \(\overline{d}\) or \(((\overline{s}_1; \overline{d})\) or \(((\overline{s}_2; \overline{d})\) or \(\ldots)))\). This requires mapping \(d\) to \(\overline{d}\), and \((f; d)\) to \(((\overline{s}_1; \overline{d})\) or \((\overline{s}_2; \overline{d})\) or \(\ldots)).\) The first point is in contrast to the first diagram above: we should not ‘pull effects out’ from \(\overline{d}\), otherwise we would produce the incorrect behaviour \(wr_{y,6}(\overline{s}_1)\) or \(((\overline{s}_1; \overline{d})\) or \(\ldots)).\) A similar observation holds if the denotation \(d\), produced by termination of the active argument \(f\) of \((f; d)\), was instead a context-term (as occurs in a transition like \(f + d \rightarrow +42(d)\)). However, we should somehow pull out effects from the \(f\) in the successor state \((f; d)\) from the non-terminal transition \(f \rightarrow f\). This illustrates the need for indicators: when evaluating a context term, one must keep track of whether each branch of the behaviour (e.g. \((f; d) \rightarrow d\) or \((f; d)\)) comes from a terminal transition \((f \rightarrow \ast)\) of the active argument, or a non-terminal transition \((f \rightarrow f)\). In the first case, if the new state is a denotation or context-term, we must not pull out any effects; whereas in the second case, we should pull them out.

A Categorical Formulation of Indicators

We now formalise the concepts of indicators and pulling out effects. To this end, we introduce a functor \(2 : C \rightarrow C\) given by \(2X = X + X\), which informally ‘decorates’ elements of \(X\) according to which component of the coproduct they are in; we understand the left component to mean the indicator is set to ‘false’ and the right to ‘true’; and we write \(\text{inf}, \text{int} : X \rightarrow 2X\) for the corresponding injections. Indicators may be discarded via the natural transformation \([\text{id}, \text{id}] : 2 \Rightarrow \text{Id}\), which we write shorthand as \([\text{id}]\).
Then we consider an operational model with indicators (for terms $TX$ over $X$) to be an $MB$- (or $B$-)coalgebra $2TX \to MB2TX$. This assigns a coalgebraic behaviour to each term $2TX$ paired with an indicator; any successor terms $2TX$ are similarly paired with indicators. However, in practise the operational models will only need to ‘set’ the indicators, to be used by the map $\hat{c}$; the operational behaviour of terms will not actually depend on the indicators. Hence it is enough to define a map $f : TX \to MB2TX$ and then take the operational model to be the coproduct $[f, f]$, written $[f] : 2TX \to MB2TX$ which discards the indicators. We will thus define operational models $\text{omdi} : TD \to MB2TD$ and $\text{omdi} : TM\overline{D} \to MB2TM\overline{D}$ for terms over denotations which incorporate these indicators, by structural recursion. We will then define a map $\hat{c} : 2TD \to B2TM\overline{D}$ which will make the following (equivalent) diagrams commute.

Now we define operational models $\text{omdi}, \text{omdi}$ for terms over denotations which incorporate these indicators, by structural recursion. For convenience, we briefly introduce an adaptation of our old notation $T^\epsilon(\gamma) : TX \to MBTX$ (Section 2.1.8) which described the operational model for terms over $X$ given by structural recursion, with base-cases given by the $MB$-coalgebra structure $\gamma : X \to MBX$. For instance, we have $\text{om} = T^\epsilon(\?_{MB0})$, $\text{omd} = T^\epsilon(\zeta)$ and $\overline{\text{omd}} = T^\epsilon(\tilde{\zeta})$. (Recall that $\zeta$ is the final $MB$-coalgebra structure, and $\tilde{\zeta} = MBu_{\overline{D}}^M \circ M\alpha^{-1}$ is our coalgebra structure for coarse-grained denotations, from Section 4.4.1).

We shortly re-use the data $r, \epsilon$ underlying the operational EIC specification $\epsilon$, to define an analogous specification $\hat{\epsilon}$ in terms of indicators. Structural recursion will then allow us to define a corresponding operational model with indicators $\hat{T}^\hat{\epsilon}(\gamma) : TX \to MB2TX$, given by the unique arrow making the following diagram commute. We then define the operational models with indicators $\text{omdi}, \text{omdi}$ over fine- and coarse-grained denotations respectively as $\text{omdi} = \hat{T}^\hat{\epsilon}(\zeta)$ and $\overline{\text{omdi}} = \hat{T}^\hat{\epsilon}(\tilde{\zeta})$. (Note that we do not need indicators in the domain $TX$ of $\hat{T}^\hat{\epsilon}(\gamma)$, as we will introduce them via the two-fold coproduct $[\hat{T}^\hat{\epsilon}(\gamma)] : 2TX \to MB2TX$. As described above, the operational model ‘ignores’, or does not depend on, the initial value of the indicator during each transition. Instead, after each transition, it will set the indicator depending on: whether the program term being executed is a base-case, redex, or context; and for context-terms, whether or not the active argument terminates.)
Here, $\hat{\epsilon}_X$ is defined analogously to $\epsilon_X$ as follows:

$$
\begin{align*}
\hat{\text{aosr}}_X : R(TX \times MB2TX) & \xrightarrow{\text{cost}_{B2TX,TX}} MH(B2TX,TX) \\
& \xrightarrow{\text{inf}_{TE}, \text{int}_{TE}} MB2TX
\end{align*}
$$

To convert from the fine-grained denotational model $D$ to the coarse-grained model $E = MD$, we define the map $\hat{c}$ as follows, where $\text{poe}$ is defined shortly. As anticipated above, $\hat{c}$ uses the map $T_{B2TX}^{\beta_\zeta}$ to converts terms over fine-grained-denotations into terms over coarse-grained denotations; but, if the indicator is set to ‘true’, it then uses the map $\text{poe}$ to pull out effects from some terms (context-terms $\sigma(t, \bar{t})$ and base-case denotations $\eta_D^T(d)$).

$$
\hat{c} : 2TD \rightarrow M2TE \quad \hat{c} = [\eta_{2TE}^M \circ \text{inf}_{TE}, \text{Mint}_{TE} \circ \text{poe}] \circ 2T_{B2TX}^{\beta_\zeta}
$$

We now define the map $\text{poe} : TE \rightarrow MTE$ by structural recursion as the unique arrow making the following diagram commute; $\text{poe}$ stands for pull out effects’ (from the active argument). The cases of reducts and contexts are handled by functions $\text{poe}^{(r)}$ and $\text{poe}^{(c)}$, defined below.
Having made all the required definitions, we will now embed the diagram (‡), from the beginning of this section, into the condition of cones as follows. It occurs at the top of the central column, and by applying $\overline{B}$ repeatedly, we obtain the other squares in that column. The squares at the top of the left and right columns (and, by applying $\overline{B}^n$, the squares below them) assert that if we discard the indicators (via the maps $[id]$), we recover the indicator-free operational models $\text{omd, omd}$. (Note the reversed horizontal arrows in the left-hand column.)

In the top-left triangle, we apply $J$ to the identity $[id] \circ \inf = [id]$. The wedge (A) amounts to the outer edges of the following commuting diagram, where the top-left area follows from the definition of $\hat{c}$, and the rest by the previous identity again, naturality of $\eta^M : Id \Rightarrow M$ or monad.
laws.

\[
\begin{array}{cccc}
TD & \xrightarrow{T\overline{\eta}} & TE & \xrightarrow{\text{id}} \downarrow \downarrow TE \\
\downarrow m_{TD} & & \downarrow \eta_{TE} & \downarrow \eta_{MTE} \\
2TD & \xrightarrow{\hat{c}} & M2TE & \xrightarrow{\mu_M} \downarrow \downarrow MTE \\
\downarrow \eta_{M2TE} & & \downarrow \eta_{M2TE} & \downarrow \mu_M \\
M2TD & \xrightarrow{\hat{c}} & M^22TE & \xrightarrow{\mu_{M2TE}} \downarrow \downarrow MTE \\
\end{array}
\]

Returning to the large diagram, the topmost thin horizontal pentagon commutes as 0 is the final object in \(\text{Kl}(M)\); applying \(\overline{B}\) to this pentagon gives the one below it – and further applications all the other horizontal pentagons. We note in passing that this fact, viz. \(M0 = 1\), is how the proof exploit the strictness of the \((\text{Cpo}_\bot\text{-enriched})\) monad \(M\), allowing us to compare the effects/comodel-manipulations of \(\text{omd}, \text{omd}\) at each execution step.

Now if we can prove the commutativity of (B)-(D), then this implies all the squares below them also commute, as they are the images of (B)-(D) in the Kleisli category under the functor \(\overline{B}\). This will ensure commutativity of the whole diagram, completing the proof of adequacy and compositionality. In the following sections, we first prove the commutativity of the squares (B) and (D), and then that of (C), which is far more difficult.

**Commutativity of (B) and (D)**

Both (B) and (D) are consequences of the following. For any \(MB\) or \(\overline{B}\)-coalgebra with carrier \(X\) and underlying structure \(\gamma : X \to MBX\), we will show that the diagram (E) in \(C\) below commutes, by structural recursion. Precomposition of (E) with \([\text{id}]\) gives diagram (F), where the left edge has replaced \(\hat{T}\gamma(\gamma) \circ [\text{id}]\) with \([\hat{T}\gamma(\gamma)]\). Using the definitions of \(J\) and of Kleisli composition, it is straightforward to show that this implies the underlying diagram in \(C\) of (B) and (D), respectively taking \(\gamma = \zeta\) and \(\gamma = \hat{\zeta}\).

\[
\begin{array}{cccc}
TX & \xrightarrow{T\gamma(\gamma)} & MBTX \\
\downarrow \hat{T}\gamma(\gamma) & & \downarrow MB[id] \\
MB2TX & \xrightarrow{MB[id]} & MBTX \\
\end{array}
\quad (E)
\]

\[
\begin{array}{cccc}
2TX & \xrightarrow{[\text{id}] \circ [\hat{T}\gamma(\gamma)]} & TX \\
\downarrow & & \downarrow \hat{T}\gamma(\gamma) \\
MB2TX & \xrightarrow{MB[id]} & MBTX \\
\end{array}
\quad (F)
\]

To prove (E), we show that \(MB[id] \circ \hat{T}\gamma(\gamma)\) fits as the unique arrow defining \(T\gamma(\gamma)\) by structural recursion. This amounts to proving the commutativity of the following diagram, (G).
The top-left and bottom-left squares are the definition of $\hat{\epsilon}_X$, and the triangles are trivial. It remains to verify the top-right square; we do this separately for each component of the coproduct $\Sigma(-)$, i.e. reducts $R(-)$ and context terms $H_2(-)$. The reduct case is given below, using definitions and the fact that $[id] \circ \inf = id$. ($\ousr_X$ is given by the left and bottom edges.)

The context case is treated in the following diagram, where the left-hand path corresponds to the abstract OS with indicators $\hat{\epsilon}_X$, and the right to the ordinary one, $\epsilon_X$. On the left-hand path, we have divided up the coproduct-components of $M\hat{\dwc}_X$, viz.

$$M\hat{\dwc}_X : M[M\inf \circ \dwc^{(v)}_X, \dwc^{(b)}_X] : M(H(V, TX) + H(2TX, TX)) \to \mathcal{M}^2 B2TX,$$

into a series of maps $M(f + g)$, and similarly on the right for the component $\dwc^{(v)}_X, \dwc^{(b)}_X : M(H(V, TX) + H_2TX) \to MBTX$ of the map $M\dwc_X$. 

\[
\begin{align*}
H(2TX, TX) & \xrightarrow{[H(\text{id}, \text{id}), H(\text{id}, \text{id})]} 2H(2TX) \\
2H(2TX) & \xrightarrow{H(2TX, TX)} H(2TX, TX)
\end{align*}
\]

Taking a coproduct of both redex and context-term diagrams gives the top-right square of
diagram (G), which implies the commutativity of (B) and (D) as anticipated.

**Commutativity of (C)**

The substantial difficulty is in proving the commutativity of square (C), viz.

\[
\begin{array}{ccc}
2TD & \xrightarrow{\hat{c}} & M2TE \\
\downarrow \text{omdi} & & \downarrow \text{omdi} \\
MB2TD & \xrightarrow{(\overline{B\tilde{c}})^\dagger} & MB2TE
\end{array}
\]

We prove it for the left and right components of the coproduct \(2TD\), both by structural recursion. The hardest case to verify occurs in both the left and right components, and will be left for the end. We start with the left component; it reduces to the left-most square in the following diagram, where the top edge is the left, or ‘false’, component of \(\hat{c}\).

\[
\begin{array}{ccc}
TD & \xrightarrow{T\beta\zeta} & TE \\
\downarrow \text{omdi} & & \downarrow \text{omdi} \\
MB2TD & \xrightarrow{(\overline{B\tilde{c}})^\dagger} & MB2TE
\end{array}
\rightarrow
\begin{array}{ccc}
TE & \xrightarrow{\text{inf}_{TE}} & 2TE \\
\downarrow \text{omdi} & & \downarrow \text{omdi} \\
MB2TD & \xrightarrow{(\overline{B\tilde{c}})^\dagger} & MB2TE
\end{array}
\rightarrow
\begin{array}{ccc}
2TE & \xrightarrow{\eta_{2TE}^M} & M2TE \\
\downarrow M\text{omdi} & & \downarrow \text{omdi} \\
M^2B2TE & \xrightarrow{M\text{omdi}^\dagger} & MB2TE
\end{array}
\]

We will prove that both paths of the reduced square, viz. \(\text{omdi} \circ T\overline{\beta\zeta}\) and \((\overline{B\tilde{c}})^\dagger \circ \text{omdi}\), serve as the unique morphism \(!\) making the following structural recursion diagram commute.

\[
\begin{array}{ccc}
\Sigma TD & \xrightarrow{\Sigma(id,\dagger)} & \Sigma(TD \times MB2TE) \\
\downarrow \psi_D & & \downarrow \Sigma(T\overline{\beta\zeta} \times \text{id}) \\
TD & \xrightarrow{\text{id}} & M2TE \\
\downarrow \eta^T_D & & \downarrow \iota_E \\
D & \xrightarrow{\overline{\beta\zeta}} & E \\
\zeta & \xrightarrow{MB\iota_E} & MBTE
\end{array}
\]

First, it is easy to show, as follows, that the diagram (I) commutes with \(! = \text{omdi} \circ T\overline{\beta\zeta}\). The left-most squares commute by naturality of \(\psi : \Sigma T \Rightarrow T\) and \(\eta^T : Id \Rightarrow T\); and the bottom-right squares commute by the definition of \(\text{omdi}\) as \(T^\zeta(\overline{\zeta})\). We shortly verify that the top-most area commutes.
The top-most area commutes by the following manipulations, where the first step uses the identity \((p \times q) \circ (f, g) = (p \circ f, q \circ g)\), the second removes and introduces some identity morphisms \(\text{id}\), and the third uses the identity \(\langle p \circ f, q \circ f \rangle = \langle p, q \rangle \circ f\).

\[
(T\bar{\beta}_\zeta \times \text{id}) \circ (\text{id}, \overline{\text{omdi}} \circ T\bar{\beta}_\zeta) = (T\bar{\beta}_\zeta \circ \text{id}, \text{id} \circ \overline{\text{omdi}} \circ T\bar{\beta}_\zeta) = \langle \text{id} \circ T\bar{\beta}_\zeta, \overline{\text{omdi}} \circ T\bar{\beta}_\zeta \rangle = \langle \text{id}, \overline{\text{omdi}} \rangle \circ T\bar{\beta}_\zeta
\]

Secondly, it remains to show that diagram (I) also commutes with \(! = (\overline{B}c)^\dag \circ \text{omdi}\). This is the hardest part of the proof. However, this task surfaces again when we consider the right component of the coproduct in (C), so we do this now. It reduces slightly into the left-most area of the following diagram. (The top edge is the ‘true’ component of the definition of \(\hat{c}\).)

\[
(T\bar{\beta}_\zeta \times \text{id}) \circ (\text{id}, \overline{\text{omdi}} \circ T\bar{\beta}_\zeta) = (T\bar{\beta}_\zeta \circ \text{id}, \text{id} \circ \overline{\text{omdi}} \circ T\bar{\beta}_\zeta) = \langle \text{id} \circ T\bar{\beta}_\zeta, \overline{\text{omdi}} \circ T\bar{\beta}_\zeta \rangle = \langle \text{id}, \overline{\text{omdi}} \rangle \circ T\bar{\beta}_\zeta
\]

Note that the bottom-left path in the left-most part, \((\overline{B}c)^\dag \circ \text{omdi}\), is the same as before, in diagram (H). Thus we aim to show both arrows again satisfy the same requirement of \(!\) in diagram (I). We start by showing the upper-right path of (H) makes diagram (I) commute, i.e. by taking \(! = \mu_{B2TE}^M \circ M\overline{\text{omdi}} \circ \text{poe} \circ T\bar{\beta}_\zeta\). First, we show it satisfies the ‘base cases’ of the structural recursion, i.e. that the bottom half of diagram (I) commutes:
From left-to-right (and bottom-to-top), the parts commute as follows: (1) $\eta^T : \text{Id} \Rightarrow T$ is natural; (2) definition of $\text{poe}$; (3) $M\eta^T : M \Rightarrow MT$ is natural; (4) $\tilde{\zeta} = MB\eta^M \circ M\alpha^{-1} = M(B\eta_D^M \circ \alpha^{-1})$, and $M\eta^M : M \Rightarrow M^2$ is natural; (5) $M$ applied to definition of $\text{omdi}$; (6) $M\eta_Y^M : MY \Rightarrow M^2Y$ is natural ($Y = D$ and $BE$); (7) monad laws; (8) $\mu : M^2 \Rightarrow M$ is natural.

Now we consider the inductive case – the upper half of diagram (H) – and reduce it to the right-most square (K) below. The top area commutes by a manipulation of products, similarly to diagram (I); the left-most area commutes as $\psi : \Sigma T \Rightarrow T$ is a natural transformation; and the middle square is from the definition of $\text{poe}$. It remains to prove the rightmost square (K) commutes, and we do this again by considering each component of the coproduct defining $\Sigma$.

The reduct case of (K) may be broken down as follows. Most of the parts of this diagram are easy to verify, using naturality of $\eta^M$ and monad laws. The large central area commutes as the left injection $\text{inl} : R \Rightarrow \Sigma$ is natural; the small square is from the definition of $\text{omdi}$; and we directly use the definition of $\text{osr}_E$ in the far-right triangle.
Similarly, the context case of (K) can be broken down into the following diagram, where the left edge is $\text{poe}^{(c)}_E$, and the right edge is $\text{aosc}_E$. The top row is a straightforward verification using products, for either side of $\times$; in the two squares below it, we use the fact that $\text{cost}'_{X,Y} : H(MX,Y) \to MH(X,Y)$ is natural in the first argument, and one of the generalised (co)strength laws from Lemma 4.2.17. Again, most of the remaining parts make use of naturality of $\mu^M$ and monad laws, in addition to: the fact that the right injection $\text{inr} : H_2 \Rightarrow \Sigma$ is natural, for the mid-left square; the definition of $\text{aosc}_E$ (like the right edge of the diagram), for the central trapezoid; and ($M$ applied to) the definition of $\text{omdi}$, for the bottom-left square.
4.4. Adequacy and Compositionality for Stateful and Effectful Languages

$$H_2(TE \times MTE) \xrightarrow{H_2(id \times M\text{omdi})} H_2(TE \times M^2B^2TE) \xrightarrow{H_2(id \times \mu_{B^2TE})} H_2(TE \times MB^2TE)$$

$$H(MTE, TE) \xrightarrow{H(\pi_2, \pi_1)} H(M^2B^2TE, TE) \xrightarrow{H(\mu_{B^2TE}, id)} H(MB^2TE, TE)$$

$$MH_2(TE) \xrightarrow{MH(\text{omdi}, id)} MH(MB^2TE, TE)$$

$$MH_2(TE) \times MB^2TE \xrightarrow{H(\pi_2, \pi_1)} MH_2(TE \times MB^2TE)$$

$$M\Sigma TE \xrightarrow{M\Sigma(id, \text{omdi})} M\Sigma(TE \times MB^2TE)$$

$$M\psi_E$$

$$M\text{omdi}$$

$$M^2B^2TE \xrightarrow{M\mu_{B^2TE}} M^2B^2TE$$
Finally, it remains to verify that the diagram (I) commutes when we set ! to the bottom-left paths of diagrams (H) and (J), viz. \((\overline{B}\hat{c})^! \circ \text{omdi}\), so that it satisfies the same universal property as the top-right parts of those diagrams. This will complete our proofs that diagrams (H) and (J) commute, and hence (C), completing the proof of the theorem. We begin with the base case, the lower part of diagram (I); it can be broken down as follows, where the arrow ! is given by the top path. (Recall that \((\overline{B}\hat{c})^! = \mu^M \circ M\lambda \circ MB\hat{c}\).)

From left-to-right and then top-to-bottom, the parts commute as follows: (1) base case of definition of \(\text{omdi}\); (2) easy consequence of definition of \(\hat{c}\) (after applying \(MB\)); (3 – large right-hand area) \(\mu_{BY}^M \circ M\lambda_Y : MBMY \to MBY\) is natural \((Y = \overline{D} \text{ and } 2TE)\); (4) \(MB\eta^r : MB \Rightarrow MBT\) is natural; (5) \(MB\) applied to base case of definition of \(\text{poe}\); (6 – bottom-left area) definition of \(\overline{\beta}\); (7) underlying definition of \((\overline{B}g)^!\) on arrows \(g\); (8 – triangle) \(MB\eta^r : MB \Rightarrow MBT\) is natural; (9 – bottom curved wedge) definition of \(\overline{\zeta}\).

The final task is to verify the inductive case, the top half of (I); just as we reduced (H) into (K), we slightly reduce this diagram into the square (M), as follows. (The left-hand square is part of the definition of \(\text{omdi}\).)

As we did for diagram (K), we verify this for each component of the coproduct defining \(\Sigma\). The reduct case is given by the following diagram whose parts commute as follows, from left-to-right and top-to-bottom: (1) \((R \text{ applied to})\) a basic property of products; (2) \(r : R \Rightarrow MBT\).
is natural; (3) \( MB\mu^T : MBT^2 \Rightarrow MBT \) is natural; (4) from the definition of \( \hat{c} \), in particular the ‘false’ component (and then applying \( MB \)); (5) one of the axioms of a distributive law \( \lambda : BM \Rightarrow MB \); (6) monad laws.

Now we come to the hardest part of the proof, which is the inductive case for context terms. We reduce the problem in two stages; first, in the following diagram (L), we reduce the problem to proving commutativity of a smaller diagram (N). The left and right edges of (L) are \( \hat{a}\text{osc}_D, \hat{a}\text{osc}_E \) respectively, and on the bottom-right we have made reference to the arrow \( B\hat{c} = (\lambda_{2TD} \circ B\hat{c}) : B2TD \rightarrow MB2TD \); it appears because the bottom edge \( (B\hat{c})^\dagger \) of (M) is equivalent to \( \mu_{B2TD} \circ MB\hat{c} \). Also note that both paths of diagram (N) end in the arrow \( M\mu_{B2TE} \); we will see this extra multiplication is required to prove its sub-cases. (There are no hats on the arrows \( dwc^{(v)}_D, dwc^{(v)}_E \); this is in accordance with the definition of \( \hat{c} \), recalling that these maps are unchanged by the introduction of indicators.)

Looking at the left half of the diagram, top-to-bottom and left-to-right the parts commute as follows: (1) properties of products; (2) definition of \( (Bf)^\dagger \) for any \( f \); (3) costrength \( \text{cost}_{X,Y} : MX \otimes Y \rightarrow M(X \otimes Y) \) is a natural transformation between bifunctors \( (X = V, Y = 2TD \text{ and } 2TE, Z = CTD) \); (4) \( \text{dist}' : (X \otimes Y) \otimes Z \rightarrow X \otimes (Y \otimes Z) \) is natural in all arguments (here, with \( X = V, Y = 2TD \text{ and } 2TE, Z = CTD \)); (5 – (M), 6 – (N)) see below; (7) \( \mu^M : M^2 \Rightarrow M \) is natural; (8) by definition of monad multiplication. The right side of the diagram is similar; the left-hand triangle is one of the laws for the generalised costrength \( \text{cost}' \), from Lemma 4.2.17.
It remains to prove (M) and (N); the latter is much more involved. In both cases, we prove the corresponding diagrams with one application of the functor \( M \) removed, and then re-apply \( M \) to get the required square. Regarding (M), as before, we prove properties of the map 
\[
\text{dist}^\prime_{X,Y,Z} : H(X + Y, Z) \to H(X, Z) + H(Y, Z)
\]
by pre- and post-composing with its inverse, 
\[
[H(\text{inl}_X, \text{id}), H(\text{inr}_Y, \text{id})] : H(X, Z) + H(Y, Z) \to H(X + Y, Z).
\]
This converts (M) into the following diagram:

\[
\begin{array}{c}
H(BM2TE, TD) \xrightarrow{H(\lambda_{2TE}, \text{id})} MH(MB2TE, TD) \\
\downarrow{[H(\text{inl}_V, \text{id}), H(\text{inr}_{M2TE}, \text{id})]} \\
H(V, TD) + H(M2TE, TD) \xrightarrow{MH(V, TD) + H(2TE, TD)} M[H(V, TD) + H(2TE, TD)] \\
\end{array}
\]

Recalling that for \( BX = V + X \), the distributive law \( \lambda_X : BMX \to MBX \) is given by 
\[
[\eta^M_{V + MX} \circ \text{inl}_V, \text{Min}_{X}] : V + MX \to M(V + X),
\]
we consider each component of the coproduct \( H(V, TD) + H(M2TE, TD) \), and have to verify equality of the outer paths of the following diagrams. The breakdown (top-down, left-to-right), is as follows. First diagram: (1) properties of coproducts; (2) one of the laws of the generalised costrength \( \text{cost}' \), in Lemma 4.2.17; (3) \( \text{cost}' \) is natural in both arguments; (4) properties of coproducts. The second diagram is similar.

Now we tackle (N), for each component of the coproduct \( H(V, TD) + H_2TD \). The left component is handled in the following (bird-shaped) diagram (O); for the right edge, recall that
The parts of diagram (R) commute as follows: looking at the left half of (R), the large leftmost trapezoid commutes as \( \text{inf} : \text{Id} \Rightarrow 2 \) is natural; and the adjacent triangle (with \( \text{inf} \)) is easily verified after postcomposing with the inverse of \( \text{dist}' \). Now the remaining parts of the left hand side commute as follows, from top to bottom: (1) functor \( H((-), \text{TD}) \) applied to fact that \( \text{inf} : \text{Id} \Rightarrow 2 \) is natural; (2 – small triangle) definition of a bifunctor \( H; \) (3) naturality of \( \text{dist} : (X + Y) \otimes Z \Rightarrow X \otimes Z + Y \otimes Z \) in all three arguments; (4) naturality of \( 2(\psi_X \circ \text{inr}_X) : 2H_2TX \Rightarrow 2TX \); (5) naturality of \( \text{inf} : \text{Id} \Rightarrow 2 \). Now the right half of (R) commutes as follows, top-bottom and left-right: (1 – small triangle) one of the properties of \( \text{cost}' \), Lemma 4.2.17; (2) \( \text{cost}'_{X,Y} : H(MX,Y) \Rightarrow MH(X,Y) \) is natural in both arguments; (3 – large right-hand area) naturality of \( \eta^M : \text{Id} \Rightarrow M \).
Lastly, we walk through diagram (S) in vertical strips. Along the left edge, the triangle is by (bi)functoriality of $H$, and the other parts commute by naturality of $\text{inr} : H_2 \Rightarrow \Sigma$ and $\psi : \Sigma T \Rightarrow T$. The next vertical strip is as follows: (1) bifunctoriality of $H$; (2) properties of products; (3) naturality of $\text{inr} : H_2 \Rightarrow \Sigma$; (4 – bottom central trapezoid) definition of $\text{poe}$. The third vertical strip: (1) naturality of $\text{cost}'$ in the right argument; (2) definition of $\text{poe}^{(c)}$; (3 – small triangle) trivial. Lastly, along the right hand side of the diagram, top-bottom and left-right: (1) naturality of $\text{cost}'$ in both arguments; (2 – small triangle) simple to verify after postcomposing with inverse of $\text{dist}'$; (3) naturality of $\text{dist}'$ in all arguments, using $\text{id}_{2TE} = 2\text{id}_{TE}$ by functoriality; (4 – large bottom-right area) naturality of $\text{Mint}_X : MX \rightarrow M2X$ ($X = H(TE,TD)$ and $TE$).
Diagram (P):

\[
\begin{align*}
H(2TD, TD) &\to H(M2TE, TD) \to MH(2TE, TD)) \to M(H(V, TD) + H(2TE, TD)) \to M(H(V, TE) + H(2TE, TE)) \\
2H2TD &\to 2H(MTE, TD) \to H(M2TE, TE) \to MH(2TE, TE) \to M[H(V, TE) + H(2TE, TE)] \\
2TD &\to \hat{c} \to M2TE \\
B2TD &\to BM2TE \to M2B2TE \\
MB2TD &\to MB\hat{c} \to MBM2TE \to M^2 B2TE \\
M\lambda_{2TE} &\to M\lambda_{2TE} \to M^2 B2TE \\
M\emptyset_{B2TE} &\to M\emptyset_{B2TE} \to MB2TE \\
\end{align*}
\]
Chapter 5

Discussion

In this chapter, we give an overview of the contributions of this thesis, and discuss possible avenues for further work.

5.1 A Mixed Kleisli Approach with Syntactic Effects

5.1.1 The Problems with Direct Application of Mathematical Operational Semantics

We began by demonstrating what can and cannot be achieved in a direct application of mathematical operational semantics to programming languages – in particular, to simple imperative programming languages with a notion of persistent store.

We showed that it is possible to describe multi-sorted program syntax $\Sigma$, and behaviour expressed in terms of $MB$-coalgebras for a functor $BX = V + X$ and monad $M$, such as the (multi-sorted) side-effect monad $MX = (S \cdot X)^S$ for a set of stores $S$. Under the assumption that an abstract operational specification $\rho_X : \Sigma(X \times MBX) \to MBTX$ may be defined, the mathematical operational semantics framework yields an operational model $om : T0 \to MBT0$ for stateful programs, a final $MB$-coalgebra morphism $\beta_{om} : T0 \to D$ into the final $MB$-coalgebra $D$ providing a characterisation of program behaviour, and a denotational semantics $[[-]] : T0 \to D$ which gives an inductive assignment of denotations $[p]$ to programs $p$. Moreover, the bialgebraic argument of Section 2.1.9 shows that these maps coincide: $\beta_{om} = [[-]]$. This implies the denotational semantics is compositional and adequate with respect to the behavioural equivalence induced by the final $MB$-coalgebra.

However, we showed that this semantics for programs suffers from two undesirable problems,
in Sections 3.1.2 and 3.1.3 respectively. The first problem is that for stateful programs like \textbf{While} – in contrast to the state-free language \textbf{SWhile} – the abstract operational specifications instantiate to very general rule formats which bear little resemblance to conventional operational rules. Standard SOS specifications for stateful languages indicate how the state changes during transition steps, for instance during variable updates, or sequential composition \( p; q \):

\[
\begin{align*}
\langle p, s \rangle & \rightarrow \langle p', s' \rangle \\
\langle p; q, s \rangle & \rightarrow \langle p'; q, s' \rangle \\
\langle p, s \rangle & \rightarrow \langle *, s' \rangle \\
\langle p; q, s \rangle & \rightarrow \langle p, s' \rangle
\end{align*}
\]

As shown, the state changes are described implicitly through syntax variables – for instance, the arrangement of the state variables \( s, s' \) above conveys the information that “the transition behaviour of \( p; q \) modifies the store in the same way as \( p \)”. We argued in Section 3.1.2 that this information needs to be made explicit if we are to provide a more restricted, and hence practical, rule format.

The second problem is that the final \( MB \)-coalgebra \( D \), characterising the behaviour of programs, contains too much fine-grained information about every execution step (Section 3.1.3). We demonstrated this in the context of stateful, \textbf{While}-like programs, by considering the construction of the final coalgebra in terms of its approximants \((MB)^n1 \) in the final sequence; we showed how these approximants imply that the final coalgebra records how every execution step modifies the state.

We then drew a parallel with the finite trace semantics of non-deterministic lts’s, where a similar phenomenon occurs. Here, a coalgebraic behaviour consists of a collection of branches – given by the power-set monad \( \mathcal{P} \) – each of which describes a labelled transition (or termination) \( BX = 1 + A \times X \) – and hence, the transition behaviour exhibited by each state corresponds to an element of \( \mathcal{P}BX \). A final \( \mathcal{P}B \)-coalgebra contains information about the branching occurring at each transition step. By contrast, given a lifting \( \overline{B} \) of \( B \) into the Kleisli category, and under suitable assumptions on \( \overline{B} \) and \( \text{Kl}(M) \), one has a final \( \overline{B} \)-coalgebra which discards this information, and characterises behaviour as a set of labelled transition traces, given by applying the monad \( \mathcal{P} \) to the initial \( B \)-algebra, which we call \( \overline{D} \).

We showed how this approach adapts into the setting of stateful programming languages, where the side effect monad \( MX = (X \times S)^S \) describes a different sort of ‘branching’ behaviour for each transition – we have one branch for each initial state of the transition, and the branches are also decorated with the final state after that transition. We replaced the labelled transition functor \( B \) with a functor \( BX = V + X \), describing either unlabelled transitions, or terminal transitions with return values given by a collection \( V \); so that the initial \( B \)-algebra \( \overline{D} = N \cdot V \) is the set of execution traces, describing the execution-time and terminal value. One has a lifting \( \overline{B} \) of \( B \), given by the distributive law of Example 2.2.10. Under suitable assumptions on \( \overline{B} \) and \( \text{Kl}(M) \), \( \overline{D} \) is the final \( \overline{B} \)-coalgebra, so that a final \( \overline{B} \)-coalgebra morphism \( \overline{\beta}_\gamma : X \rightarrow \overline{D} \)
has underlying type $X \to M\overline{D}$. This amounts to changing our semantic domain for programs from the final $MB$-coalgebra $D$ into $M\overline{D}$. We interpreted the semantic domain $M\overline{D}$ in terms of state-manipulations: each program is assigned a function which, for each initial state in $S$, returns the final state $S$ along with the number of steps-to-termination and the return value of that program.

We showed that the corresponding approximants $M(B^n)0$ to the semantic domain $M\overline{D}$ improve on the fine-grainedness of the final $MB$-coalgebra, by ‘chaining together’ the state-manipulations at each step; this discards information about the state changes at each individual step, leaving only the overall state-change observed during execution.

### 5.1.2 Introducing the Mixed Kleisli Setting

We then showed how the original categorical setting had to be modified to accommodate a final-coalgebra semantics in the Kleisli category (Section 3.2). The difficulty is that one may not define polynomial syntax functors in the Kleisli category $Kl(M)$, due to the absence of categorical products in general; thus, one may not simply take $Kl(M)$ as the categorical setting for mathematical operational semantics. We argued that the problem arises because syntax does not naturally belong in the Kleisli category; the composition of Kleisli-arrows is intimately tied up with the propagation of state manipulations (or the accumulation of effects), whereas syntax functors are mainly concerned with relabelling or substituting arguments.

This motivated our decision to define program syntax $T0$ in the underlying category, and hence also a denotational map $[\cdot] : T0 \to M\overline{D}$ (assuming one has defined a $\Sigma$-algebra structure $\overline{dm} : \Sigma M\overline{D} \to M\overline{D}$). By contrast, the operational model $om : T0 \to MBT0$ is equivalent to a $\overline{B}$-coalgebra in the Kleisli category, and the characterisation of behaviour $\overline{\beta}_{om} : T0 \to \overline{D}$, in terms of the final $\overline{B}$-coalgebra, also takes place in the Kleisli category. The syntactic and behavioural descriptions are connected by the fact that the denotational and operational maps $[\cdot], \overline{\beta}_{om}$ are essentially of the same type $T0 \to M\overline{D}$, assigning each program an element of the semantic domain; as in the original setting, the problem of proving adequacy and compositionality amounts to proving equivalence of these two arrows.

To make progress on this problem, and to motivate later developments, we begin in more concrete terms by restricting attention to syntactic effects in Chapter 3. In this perspective, instead of describing state-manipulations, program execution branches according to external factors, such as the value of a variable in the store, or a non-deterministic choice. We represented these branchings, or effects, over executions in the same way as we described program syntax, by free $\Delta$-algebras $T_eX$ for suitable syntax functors $\Delta$ – so that the semantic domain $T_e\overline{D}$ becomes the collection of syntactic effect-trees, with leaves describing individual execution-paths in terms of the steps-to-termination and return value.
5.1.3 Order-Enrichment for a Semantic Domain

Having defined our semantic domain $T_e\mathcal{D}$ in terms of syntactic effects and an initial $B$-algebra $\mathcal{D}$, we addressed the problem of ensuring that $\mathcal{D}$ is indeed a final $\mathcal{B}$-coalgebra, so that one has a $\mathcal{B}$-coalgebra morphism $\beta_{\text{om}} : T_0 \to \mathcal{D}$ of underlying type $T_0 \to M\mathcal{D}$, characterising the behaviour of program states. To do this, we made use of existing results on limit-colimit coincidences in $\mathcal{Cpo}_{\perp!}$-enriched categories; we showed that by taking the base category to be $\mathcal{Cpo}_{\perp!}$, the initial $B$-algebra $\mathcal{D}$ is indeed a final $\mathcal{B}$-coalgebra if the monad $M$ is strict (Corollary 3.3.3).

We described how this order-enrichment affects the semantic domain, introducing divergent leaves $\perp$ and an order-structure to the syntactic effect-trees $T_e\mathcal{D}$. We then showed how syntax functors in $\mathcal{Cpo}_{\perp!}$ induce this order structure on both program-syntax terms and effect-trees, by discussing the initial algebras of polynomial functors $\Sigma, \Delta$ incorporating both smash and Cartesian products, which may be guaranteed to exist by exploiting the property of algebraic $\omega$-compactness. We gave a criterion for verifying the concrete structure of these initial algebras (Remark 3.3.8), and proved that the free $\Delta$-algebra monad $T_e$ is strict (Lemma 3.3.11) in the absence of constants (i.e. nullary effects); this ensures that $T_e\mathcal{D}$ may serve as a semantic domain for programs with syntactic effects.

In the absence of an established theory of bisimulations for such programming languages, we gave a criterion for checking properties of the final $\mathcal{B}$-coalgebra morphism $\beta_{\text{om}} : T_0 \to T_e\mathcal{D}$ into the final $\mathcal{B}$-coalgebra, through a least-fixpoint construction (Proposition 2.2.18); we showed how this least-fixpoint definition gives a concrete characterisation of program behaviour, in Example 3.3.12.

5.1.4 Operational Specifications for Syntactic Effects

We had already discussed the difficulties in defining operational specifications $\epsilon$ for stateful languages in Section 3.1.2; by contrast, many language constructs essentially do not interact with effects, and these are easily specified in an effect-free setting by natural transformations $\rho : \Sigma(X \times BX) \to BTX$ – where $\Sigma$ and $T$ refer to the syntax constructors of the effect-free fragment of the language. We illustrated such a specification for if statements (Example 3.1.7) in the effect-free language $\text{SWhile}$ of While. This observation served as a starting point for a more refined approach to operational specifications for languages with effects, as we addressed the question of how to extend an effect-free specification $\rho$ to incorporate effects (Section 3.3.4).

To derive the effectful behaviour of a syntax term, the main intuition was that one must apply the effect-free specification – i.e. effect-free operational rules – ‘at every branch of the effectful behaviour’ of certain, active arguments. These active arguments are only implicitly identified
in operational rules, in the same way that stateful operational rules, such as those for sequential composition \( p; q \) above, implicitly propagate state-manipulations from selected arguments, such as the \( p \) in \( p; q \). This leads us to the definition of ‘dependency functions’, identifying the active arguments of each syntax constructor.

On a formal level, monadic strength (of the effect-syntax monad \( T_e \)) is used to attach the relevant information to each branch of these behaviours. However, the setting of \( \text{Cpo}_\bot \) introduces some subtle restrictions on the syntax functor \( \Sigma \) about the use of smash (or monoidal) products, and Cartesian (or categorical) products; monadic strength cannot be adapted to handle Cartesian products. Hence, monoidal products must be used for active arguments, to permit the use of monoidal strength to manipulate effect-trees; Cartesian products are needed for the rest, so that we can define specifications in terms of categorical projections (as suitable analogues do not exist for the smash product). There is also a similar restriction on the effect-free specifications \( \rho \): they may not refer to the ‘current’ values of the active arguments (such as the \( p \) in \( p; q \)), but only their successor or termination value (e.g. on the \( p' \) in \( p \rightarrow p' \), or the value \( * \) in \( p \rightarrow * \)).

Subject to these restrictions on the syntax functor \( \Sigma \) and effect-free specification \( \rho \), we showed how the latter may be extended to specify interactions between the effect-free fragment of the language and the newly-introduced effects. This gives a uniform means of deriving operational specifications \( \epsilon \) for effectful languages, and hence an operational model \( \text{om} : T0 \rightarrow MBT0 \), from simpler, effect-free specifications.

### 5.1.5 Adequacy and Compositionality for Syntactic Effects

In contrast to the assignment of behaviour to programs \( \overline{\beta}_{\text{om}} : T0 \rightarrow M\overline{D} \), one seeks a denotational semantics, an inductive assignment of meanings \( [p] \) to programs \( p \); this requires making the semantic domain \( M\overline{D} \) into a denotational model (i.e. a \( \Sigma \)-algebra) with structure \( \overline{dm} \), which we referred to as ‘coarse-grained’. We achieved this by following similar steps to the ones making the final \( MB \)-coalgebra \( D \) into a denotational model \( dm \), referred to as ‘fine-grained’. These steps involved introducing ‘operational models over denotations’, which we called \( \text{omd} \) and \( \text{omd} \) respectively; we illustrated their behaviour with some examples. The key difference is that, given a program \( p \), its fine-grained denotation introduces the effects of \( p \) gradually, whereas its coarse-grained denotation introduces them all at once (as this denotation has discarded the information about precisely when those effects occurred).

We then considered how to prove adequacy and compositionality of the coarse-grained denotational semantics \( [-] \) with respect to the behavioural equivalence induced by the final \( \overline{\beta} \)-coalgebra map, \( \overline{\beta}_{\text{om}} \), by showing that these maps coincide. We gave some examples to illustrate that not all syntax constructors give rise to adequate semantics in this way, and that the class
of abstract operational specifications needed to be specialised: for this purpose, we introduced a restricted class called the \textit{effectfully extended Evaluation-in-Context} (eEIC) Format, defined in terms of the effectful extensions introduced above. The key idea was to restrict attention to either context terms, where a single active argument determines the effects exhibited by the term; or redex terms without any active arguments, whose transition behaviour is independent of the immediate behaviours of its arguments.

Assuming this restriction on abstract operational specifications, to show that the maps \([-], \beta\ om\) coincide, we exploited the fact that the ‘fine-grained’ semantic maps \([-], \beta\ om : T0 \rightarrow D\), obtained under the original framework, are the same: in particular, they both satisfy the initiality property of the initial \(\Sigma\)-algebra morphism \(T0 \rightarrow D\). We used this fact to reduce the problem into showing that the canonical \(\overline{B}\)-coalgebra map \(\overline{\beta}_\zeta : D \rightarrow M\overline{D}\) between the semantic domains is also a \(\Sigma\)-algebra morphism. We described how this map may be thought of as ‘coarsening’ the denotations by ignoring the precise execution steps at which effects are introduced, and only retaining the execution-time and return value of each computation branch.

By phrasing \(\overline{B}\)-coalgebra morphisms in terms of cone morphisms, we re-expressed this requirement as the commutativity of a large diagram, and we gave a syntactic proof that it commuted, by relating the effect-trees produced by the transitions of the operational models over denotations \(\text{omd, omd}\). At the heart of this proof was the idea that the syntactic effects were \textit{strict}, i.e. that any all-\(\perp\) effect-trees \(\delta((\perp)_{i \in I})\) are identified with \(\perp\); as a result, even though the operational model \(\text{omd}\) introduces ‘more effects’ than \(\text{om}\) at each transition step, the leaves of those effects are sent to \(\perp\) by the arrows in the diagram, and hence these extra effects may be ignored. This allows us to make a correspondence between the effectful behaviour of the operational models at each execution step. We re-used this idea as the guiding principle in a fully categorical proof of adequacy and compositionality, in Chapter 4.

\section{5.2 Introducing Comodels and Semantic Effects}

At this point, we had outlined an adaptation of the original mathematical operational semantics into the setting of programs with syntactic effects. The definitions were given with respect to the syntactic effect-monad \(T_e\), in particular in the operational model \(\text{om} : T0 \rightarrow T_eBT0\) and the semantic domain \(T_e\overline{D} = T_e(N\cdot V)\). On a formal level, it is a simple matter to replace the monad \(T_e\) with an arbitrary monad \(M\); this means considering operational models \(\text{om} : T0 \rightarrow MBT0\) as \(MB\)-coalgebras, and corresponding \(MB\)-coalgebra specifications by extending effect-free specifications \(\rho\) as described before. As before, Example 2.2.10 shows there is a lifting \(\overline{B}\) of \(B\) into the Kleisli category \(\mathcal{Kl}(M)\); providing the monad \(M\) is strict, there is also a final \(\overline{B}\)-coalgebra \(\overline{D} = N \cdot V\), so that the final \(\overline{B}\)-coalgebra morphisms \(\overline{\beta}_\gamma : X \rightarrow M\overline{D}\) allow us to take \(M\overline{D}\) as a semantic domain. The problem of proving adequacy and compositionality reduces to
commutativity of the same diagram as before (which we anticipated by replacing $T_e$ with $M$ throughout that section), which must now be done categorically rather than syntactically.

Although this generalisation is simple on a formal level, it raises several important issues that need to be addressed. The first is whether or not this abstract approach instantiates into a meaningful coalgebraic semantics for programming languages with effects given by a Lawvere theory $L$, or a notion of state given by a comodel $C : L \to \mathbb{Cpo}_{\bot}$. We showed that it does, by introducing several new classes of transition systems: instead of considering program execution in terms of syntactic effects – what we called syntactic effectful transition systems (ETS’s) – we introduced a notion of behaviour in terms of models of Lawvere theories, in which these syntactic effects are essentially quotiented by an algebraic theory; this gives rise to the class of semantic ETS’s. Given a comodel $C$ of a Lawvere theory $L$, we also introduced a notion of behaviour related to the side-effect monad, where the ‘state’ $S$ is given by the state-space $C1$ of the comodel. Finally, we considered languages with a more elaborate interaction between comodels and effects, where the comodel is for a sub-theory $L_1$ (such as global store) of a theory $L = L_1 \otimes L_2$ given by a tensor product with another theory $L_2$ (such as non-determinism); these correspond to the class of comodel and effect-based transition systems (CETS’s), which may exhibit either syntactic effects (syntactic CETS’s) or their equivalence classes (semantic CETS’s).

We showed how these classes of transition systems could be represented by $MB$-coalgebras for suitable choices of the monad $M$. We may take it to be the monad $N_e$ giving carriers of free models of $L$ – which gives rise to the class of semantic ETS’s – or the side-effect monad $N_c$, where the ‘state’ $S$ is given by the carrier $C1$ of the comodel; and this gives coalgebras which are equivalent to CTS’s. Suitable monads $T_{ce}, N_{ce}$ give coalgebraic representations of CETS’s.

However, merely providing these coalgebraic descriptions does not address the role played by the comodel-structure, and whether there is any connection between the effect-based and comodel-based semantics – in particular, whether effects may ‘flow’ between the program and the comodel, as described in [PP08]. Moreover, we have referred informally to ‘equivalence classes’ of effects, given by quotienting effect syntax, and the question arises about whether this quotient may be expressed formally. We addressed these questions by providing monad morphisms between the various monads used in these coalgebraic descriptions. We formalise the process of passing effects to a comodel, via a monad morphism $m^c : N_e \Rightarrow N_c$, and similarly $m^{ce} : N_e \Rightarrow N_{ce}$ to describe this process for CETS’s, where we can only pass effects from a sub-theory $L_1$ into the comodel. Finally, we gave a monad morphism $quot : T_e \Rightarrow N_e$ which formalises the ‘quotienting’ of effect-syntax into a free model of the Lawvere theory.
5.2. Introducing Comodels and Semantic Effects

5.2.1 Operational Specifications

Having introduced the above classes of transition systems, the second major issue to address was how to specify operational models. Our discussion of While in Section 3.1.2 had already demonstrated the impracticality of abstract operational semantics for MB-coalgebras, and moreover we had introduced the eEIC format to ensure adequacy and compositionality of our coarse-grained denotational and operational semantics. However, this format was defined syntactically in terms of effectful extensions, and it is not immediately clear how it relates to the conventional specifications for stateful languages such as While.

For these reasons, we gave concrete definitions of evaluation-in-context rule formats (EIC1-3) for each class of transition system; in particular, we produced a natural rule format (EIC1) for stateful languages, with operational models given by cts’s. The concrete formats for (c)ets’s also offer some intuition about how program behaviour is derived by structural recursion, given EIC specifications for these transition systems; this helped us in our search for a more general proof of adequacy and compositionality.

For each concrete rule format, we gave a corresponding categorical interpretation, and showed how the specification data may be formalised by monadic EIC specifications, instantiated for different monads $M$. This involved restricting the syntax functor $\Sigma$ to make explicit references to multi-sorted redexes $RX$ and context terms $H_2(X) = H(X, X)$, where the first argument of $H$ is the active argument. We also generalised the costrength $\textsf{cost}$ and distributivity $\textsf{dist}$ to extract effects from multi-sorted context terms. These definitions were used to show how a monadic EIC specification in turn defines an abstract operational specification $\epsilon_X : \Sigma(TX \times MBTX) \to MBTX$, allowing us to induce operational models by structural recursion in a uniform manner for each class of transition system.

5.2.2 Semantic Domains

Given an operational model, the next question to ask is whether one has a final $\overline{B}$-coalgebra $\overline{D}$ in the Kleisli category $\text{Kl}(M)$, so that the codomain $MD\overline{D}$ of the final coalgebra morphisms $\overline{\beta}_\gamma : X \to MD\overline{D}$ may serve as a semantic domain for programs. This required proving strictness of the monads $M$ used to define the transition systems; and $\text{Cpo}_{\perp \perp}$-enrichment of $M$ is sufficient for this. We achieved this for each monad corresponding to a class of transition systems; the main difficulty was in proving that the free-model monad $N_e$ is $\text{Cpo}_{\perp \perp}$-enriched. To do this, we had to introduce the enriched Lawvere theory $\mathcal{L}'$ freely generated by an ordinary theory $\mathcal{L}$. The enriched theory induces a $\text{Cpo}_{\perp \perp}$-monad $N'_e$ whose underlying ordinary monad coincides with the monad $N_e$ induced by the latter, so that we may consider $N_e$ to have the required enrichment.
Having ensured existence of the semantic domains $M\overline{D}$, we gave concrete descriptions of the induced behavioural equivalences $\cong$ on programs, whereby two programs $p, q$ are identified iff $\overline{\beta}_{om}(p) = \overline{\beta}_{om}(q)$. In the case of the cts, the equivalence $\cong_e$ identifies two programs if and only if for every initial comodel-state, the programs terminate with the same final comodel-state, return value, and execution-time; the intermediate state-manipulations are irrelevant. Similarly for the classes (c)ets of transition systems with effects, the characterisations of programs $\cong^N_e$ and $\cong^N_{ce}$ do not depend on the effects occurring at each execution step, but only on the overall effect-tree accumulated during execution. This demonstrates that this semantic domain resolves the problem of excessive fine-grainedness which we described in the original semantic domain of the final $MB$-coalgebra.

These behavioural equivalences, induced by the final coalgebra morphisms, are suitable for describing most of the transition systems considered, but it is not entirely satisfactory for the transition systems which are defined by syntactic effects (i.e. syntactic (c)ets’s). Their equivalences $\cong^T_e$ and $\cong^T_{ce}$ identify states of these transition systems iff they produce exactly the same effect-trees; whereas one would not wish to distinguish effect-trees which are identified by the corresponding algebraic theory. To address this problem, we introduced behavioural equivalences as follows: after applying the coalgebra morphisms $\overline{\beta}_{om} : T0 \rightarrow M\overline{D}$ into the semantic domain, we post-compose with suitable quotienting maps, derived from the monad morphism quot, which send each effect-tree to its equivalence class under the equational theory. We then obtain more satisfactory behavioural equivalences $\cong^{T \rightarrow N}_e$ and $\cong^{T \rightarrow N}_{ce}$, where programs are identified iff they produce the same equivalence class of effect-trees, with leaves labelled by the execution-time and return value of that computation branch (or divergence $\perp$).

### 5.2.3 A Categorical Proof of Adequacy and Compositionality

In the same way as before, we equipped the semantic domain $M\overline{D}$ with a $\Sigma$-algebra structure $\overline{dm}$, so that the initial $\Sigma$-algebra maps $\llbracket - \rrbracket : T0 \rightarrow M\overline{D}$ provide a denotational semantics for programs. For each class of transition system, we gave concrete examples of the interpretations of syntax constructors on denotations.

For most of the transition systems we considered, behavioural equivalence was characterised by the final $\overline{B}$-coalgebra morphisms $\overline{\beta}_{om} : T0 \rightarrow M\overline{D}$; hence the problem of proving adequacy and compositionality again reduced to showing that $\overline{\beta}_{om}$ is a $\Sigma$-algebra morphism into the denotational model. This was proven in Theorem 4.4.4, by appealing to the same diagram as before, given by a morphism of cones. The difference is that a monadic EIC specification now allows us to make the proof fully categorical. However, the underlying principal is the same: the strictness of the monad $M$ allows us to match the effect-trees and/or comodel manipulations produced at each transition step by both operational models $\overline{omd, omd}$ over fine- and coarse-
grained denotations; the non-terminal leaves of these effect-trees are then sent to ⊥, so that the all-⊥ subtrees are removed, and the effect-trees are identified. The difficulty in the proof is that this matching needs to extract effects/comodel-changes from context terms σ(x, ˜x) in different ways depending on whether the execution branches of the active argument x terminate or not; and this requires the introduction of binary indicators.

This proof yields adequacy and compositionality of the semantics for the transition systems we have considered, with respect to the behavioural equivalences characterised by final \( \mathcal{B} \)-coalgebra morphisms. The remaining behavioural equivalences \( \cong_{\mathcal{E} \rightarrow N} \) and \( \cong_{\mathcal{C} \rightarrow N} \) – for syntactic \((C)ETS's\) – were characterised by post-composing the (now \( M \)-decorated) final coalgebra maps \( \beta^M_{\text{om}} \) with suitable monad morphisms \( m: M \Rightarrow N \) for quotienting out the effect-trees; to handle those cases, we instead had to show that \( m_{\mathcal{T}} \circ \beta^M_{\text{om}} \) was a \( \Sigma \)-algebra morphism. We did this indirectly, by showing that our semantic framework is in a sense ‘parametric’ in the monad \( M \); one may use a monad morphism to translate features, such as the denotations in the semantic domain \( MD \), into analogous features in terms of \( N \). We used this fact to show that the composition \( m_{\mathcal{T}} \circ \beta^M_{\text{om}} \) – assigning denotations in the semantic domain \( MD \), and then translating them into \( ND \) – is equivalent to the final coalgebra map \( \beta^N_{\text{om}} \) in \( \text{Kl}(N) \), which instead assigns denotations directly in \( ND \), from a ‘translated’ operational model \( \text{omt}: T0 \rightarrow NBT0 \). Theorem 4.4.4, instantiated with the monad \( N \) rather than \( M \), then shows that \( m_{\mathcal{T}} \circ \beta^M_{\text{om}} = \beta^N_{\text{om}} \) is a \( \Sigma \)-algebra morphism, completing the proof of adequacy and compositionality for the remaining behavioural equivalences.

5.3 Applications, Adaptations and Limitations

We demonstrated our methods by specifying operational models for several example languages, and gave a concrete description of the resulting semantic domains. These included: the While language, represented in terms of the canonical comodel for global store; a variant \( \text{NDWhile} \) with non-deterministic branching, in addition to comodel-dependent executions; and various extensions of an ‘effect-free’ base language \( \text{SWhile} \) with syntactic effects for non-determinism and/or global store. Other possible applications include languages with probabilistic or graded non-determinism, and models of local store in a suitable presheaf over \( \text{Cpo}_{\perp!} \) (or ‘nominal \( \omega \)-Cpos’), along the lines of [PP02]. More speculatively, by working in (a suitable presheaf over) the mixed-variance (but still \( \text{Cpo}_{\perp!} \)-enriched) setting of \( \text{Cpo}^m_{\perp!} \times \text{Cpo}_{\perp!} \), one may be able to express the behaviour of functional languages coalgebraically, and exploit a limit-colimit coincidence to ensure existence of a semantic domain. However, even in the absence of effects or comodels, one would have to address questions about the most natural way to model syntax and behaviour in this setting, and the interpretation of mixed-variance abstract OS specifications.

The framework may also be used to model interactive I/O, by a syntactic ETS; however, the
resulting treatment of divergence is unsatisfactory. The monads in our framework (in particular, \( T_e \)) have to be strict, and as described in Section 3.3.3, this means that any syntactic effect-trees \( \delta \) have to satisfy \( \delta((\bot)_{i \in I}) = \bot \). Thus, a program \( p \) which produces output ‘a’ and then diverges receives the denotation \( [p] = \text{out}_a(\bot) = \bot \), and so \( p \) would be considered behaviourally equivalent to a program which immediately diverged: the intermediate output ‘a’ is ignored by the semantics. Nonetheless, the semantics does record any I/O activity arising from terminating executions.

### 5.3.1 Non-strict Monads

One may attempt to relax the strictness requirement on monads \( M \), as follows. Firstly, rather than \( \text{Cpo}_{\bot} \), one may take the base category \( \text{Cpo}_{\bot} \) given by non-strict \( \omega \)-continuous maps between pointed \( \omega \)-cpos, and require the monad \( M \), and hence the Kleisli category \( \text{Kl}(M) \), to be \( \text{Cpo}_{\bot} \)-enriched. Alternatively, one may work in the category \( \text{Cpo} \) of non-pointed \( \omega \)-cpos, and ‘introduce divergence within the monad \( M \’ \) (say, by a nullary operator \( \bot \) in a discrete Lawvere theory) ensuring that \( \text{Kl}(M) \) is again \( \text{Cpo}_{\bot} \)-enriched – and not necessarily with left-strict composition (which, adding the easier property of right-strictness, would amount to the strict \( \text{Cpo}_{\bot} \)-enrichment we wish to avoid).

Under this assumption, the semantic domain \( M\overline{D} \) in the Kleisli category \( \text{Kl}(M) \) becomes a weakly final \( \mathcal{B} \)-coalgebra; but one retains a canonical description of the coalgebra morphisms \( \overline{\beta}_{om} : T0 \to \overline{D} \), given by the least-fixpoint construction of Proposition 2.2.18, as they are the least such coalgebra morphisms in the ordering on \( \text{Cpo}_{\bot} \)-arrows. This property allows us to rephrase the condition of cones in Section 3.4.3 into a very similar criterion for proving adequacy and compositionality.

However, one finds that without strictness, the proof of adequacy and compositionality becomes considerably harder. We relied on strictness to ‘locally’ match the effects introduced by a single step of both operational models \( \text{omd}, \overline{\text{omd}} \) over fine- and coarse-grained denotations. Without strictness, it seems the only available strategy is essentially to match the effects produced by \( m \) steps of the coarse-grained model \( \overline{\text{omd}} \), with the effects produced by \( mn \) steps of the fine-grained model \( \text{omd} \), for all \( m, n < \omega \). At the time of writing, we were unable to produce such a proof, partly because of the complexity of the matching; it may be possible to achieve this in future, but it may instead be more fruitful to pursue the alternative approaches outlined below.

### 5.3.2 The Number of Steps-to-Termination

A related criticism of our framework is that the semantic domain records the number of steps-to-termination of each execution branch; thus, programs like \( (x = 1) \) and \( (x = 1; \text{skip}) \) are
considered to have different behaviour. Note that this also occurs in the original, fine-grained semantics, as given by the final $MB$-coalgebra. In general, abstracting away the number of steps-to-termination corresponds to a form of weak coalgebraic bisimilarity, for which no general methods are known. However, if the strictness requirement on monads $M$ may be relaxed, then our framework would have the potential to overcome this limitation. This is because the behaviour functor $BX = V + X$ is itself a monad, with multiplication $[\text{inl}, [\text{inl}, \text{inr}]] : V + (V + X) \to V + X$ which ‘forgets’ the distinction between one-step and two-step behaviour; and the distributive law $\lambda : BM \Rightarrow MB$ of Example 2.2.10 makes the composition $MB$ into a non-strict monad. Rather than lifting the functor $B$ into a functor $\overline{B}$ on $\text{Kl}(M)$ and construing $MB$-coalgebras as $\overline{B}$-coalgebras, we instead (trivially) lift the identity functor $\text{Id}$ and construe them as $\text{Id}$-coalgebras in $\text{Kl}(MB)$. As the initial $\text{Id}$-algebra is the initial object $0$, the semantic domain $MD$ is replaced with $(MB)0 \cong MV$, which contains no reference to the number of steps-to-termination, and only return values. One obtains a condition of cones closely related to the one we would obtain under a non-strict semantics; and so a proof of adequacy and compositionality for non-strict monads $M$ would allow us to discard the execution-time of programs.

### 5.3.3 Working in a Category of Algebras

As an alternative to our semantic framework in a Kleisli category, one may attempt to apply mathematical operational semantics directly in the category $\text{EM}(M)$ of Eilenberg-Moore algebras\footnote{This is essentially the category $\text{Alg}(M)$ of algebras $A = (X, \gamma : MX \to X)$ for the functor $M$, with additional constraints. One may see the Kleisli category $\text{Kl}(M)$ as a subcategory of $\text{EM}(M)$, corresponding to the free $M$-algebras.} for the monads $M$ of interest, on some base category $C$ (not necessarily $\text{Cpo}_\bot$). The potential advantages of this approach are that (1) it is easier to guarantee the existence of final coalgebras in $\text{EM}(M)$ than it is in $\text{Kl}(M)$; and (2) we would not need to provide a proof of adequacy and compositionality, as the original bialgebraic proof would still stand.

We briefly outline how one may attempt to pursue this approach. In analogy to the adjunction $J \dashv L$ relating the underlying and Kleisli categories $C, \text{Kl}(M)$ (Definition 2.2.3), one now requires the existence of an adjunction $F \dashv U$, where $U : \text{EM}(M) \to C$ is the forgetful functor mapping an algebra $A = (X, \gamma : MX \to X)$ to its carrier $UA = X$. Its left adjoint $F : C \to \text{EM}(M)$ can be thought of as the free $M$-algebra functor. Given an object $V$ in $C$ representing a collection of terminal values, one may then define a behaviour functor $BA = FV + A$ on $\text{EM}(M)$; this coproduct of algebras intuitively constructs (suitable equivalence classes of) effect-trees, with leaves labelled either by values $v \in V$ or elements $a$ of the carrier $UA$ of the algebra $A$. Thus, a $B$-coalgebra $A \to FV + A$ would play the part that $MB$-coalgebras $X \to MBX$ played in our framework: it assigns effectful behaviours to states – which now possess an $M$-algebra.
structure, essentially interpreting the effect-operations on states.

However, such an approach faces a new set of challenges. One characteristic of our mixed-Kleisli setting is the clear separation of syntax and semantics, expressed by separate functors $\Sigma$ and $M$ respectively; and this distinction is lost if one attempts to represent a syntax functor in $\text{EM}(M)$ directly. For instance, if $M = T_e$ is the free effect-syntax monad, then one is forced to provide interpretations of effect-operators on the syntax terms of the language; and there are several non-canonical ways of providing these interpretations on syntax terms.

The conflation of syntax and semantics also makes it more difficult to define and interpret abstract operational specifications. Our mixed-Kleisli approach defined these specifications $\epsilon$ as natural transformations in the underlying category; this made it possible to be concrete about the mechanics of EIC specifications, in terms of (co)products, distributivity, and monadic (co)strength. However, to define specifications categorically in terms of (co)products, one requires distributivity of coproducts over products; and syntactic examples suggest that it is unlikely this distributivity will exist in general, unless the monad $M$ is commutative – an assumption we have deliberately tried to avoid, as it excludes many monads of interest in program semantics.

The third difficulty is that although final coalgebras exist for most functors $\hat{B}$ on $\text{EM}(M)$ of interest, in general their structure is hard to describe concretely. Existing results have focused on cases where $\hat{B}$ is a (correctly named!) lifting of some functor $B$ (i.e. satisfying $U \hat{B} = BU$); whereas the behaviour functor $BX = V + X$ only has an extension $\hat{B}A = FV + A$ (i.e. satisfying $FB = \hat{BF}$) – see e.g. [JSS12, BK10]. Alternatively, one may be able to develop a concrete description of this final coalgebra in terms of equational theories in order-enriched categories [AJ94, Rob02].

## 5.4 Conclusion

The starting point for this thesis was the question of whether or not mathematical operational semantics can be applied to programming languages; and we have at least partially showed that the answer is ‘yes’. We adapted the framework by drawing on methods of coalgebraic trace semantics, and results from existing research into computational effects, to give a better characterisation of program behaviour than was possible in the original setting. In the process, the framework makes a connection between the operational semantics of programming languages, and the theory of computational effects and comodels. Regarding its applications, in this section we have outlined how future work may broaden the class of examples covered by the framework, and moreover optionally discard the execution-time of programs, if the assumption of strictness can be relaxed.
However, the elegance of the original categorical framework does not extend into the class of languages we have considered; our central proofs of adequacy and compositionality involve a lot of detail, and are difficult to generalise in the ways we outlined above. One may see this as a consequence of the fact that these properties require careful restrictions on operational specifications, which are not easily expressed in categorical terms.

There may be an alternative solution, given by the category of algebras $\text{EM}(M)$. Here, the difficulties of categorical definitions essentially ‘rule out’ many operational specifications; and this may enforce the required restrictions implicitly, rather than explicitly as we have done. We have described some of the new questions which would have to be tackled in this approach; however, we believe that those questions will be made easier to answer by our investigation, as we have given concrete interpretations, and hopefully insights, into the constructions which are often hidden beneath the relevant categorical definitions – and these will become increasingly prominent in a category of algebras.

Our work has not provided a definitive solution to the problems posed by applying mathematical operational semantics in the context of programming languages; but we have made a detailed investigation into the possibilities and limitations of this approach, and we have laid some groundwork for further attempts to explore the potential of coalgebraic semantics for programming languages.
Bibliography


