Technical Report

UCAM-CL-TR-854
ISSN 1476-2986

Number 854

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Programming contextual computations

Dominic Orchard

May 2014

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted January 2013 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Jesus College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Modern computer programs are executed in a variety of different *contexts*: on servers, handheld devices, graphics cards, and across distributed environments, to name a few. Understanding a program's *contextual* requirements is therefore vital for its correct execution. This dissertation studies contextual computations, ranging from application-level notions of context to lower-level notions of context prevalent in common programming tasks. It makes contributions in three areas: mathematically structuring contextual computations, analysing contextual program properties, and designing languages to facilitate contextual programming.

Firstly, existing work which mathematically structures contextual computations using *comonads* (in programming and semantics) is analysed and extended. Comonads are shown to exhibit a *shape preservation* property which restricts their applicability to a subset of contextual computations. Subsequently, novel generalisations of comonads are developed, including the notion of an *indexed comonad*, relaxing shape-preservation restrictions.

Secondly, a general class of static analyses called *coeffect* systems is introduced to describe the propagation of contextual requirements throughout a program. Indexed comonads, with some additional structure, are shown to provide a semantics for languages whose contextual properties are captured by a coeffect analysis.

Finally, language constructs are presented to ease the programming of contextual computations. The benefits of these language features, the mathematical structuring, and coeffect systems are demonstrated by a language for container programming which guarantees optimisations and safety invariants.

Acknowledgements

There are many people who I owe a debt of gratitude for their help, support, and encouragement during my PhD.

I am extremely grateful to my supervisor, Alan Mycroft, for his endless enthusiasm, patience, wisdom, and support.

I am indebted to my PhD examiners, Marcelo Fiore and Tarmo Uustalu, for their thoughtful comments and questions, which have greatly improved this dissertation. Any remaining errors and infelicities are my own.

Thank you to the Computer Laboratory, Jesus College, the Cambridge Programming Research Group, and the Programming, Logic, and Semantics group, for friendship and a great working environment. In particular thanks to Robin, Kathy, Max, Pete, Jukka, Tomas, Raoul, and Janina for shared enjoyment in programming language research and many burritos. Particular thanks to Tomas Petricek for his encouragement, friendship, and many conversations about comonads!

Thank you to all those who I have visited, or had conversations with (usually in a pub) over the years. You are too numerous to name. Particular thanks to Tarmo Uustalu and Varmo Vene for inspiring me.

Thank you to all my friends and family who have been immensely supportive and loving; particularly my mother, father, sister, Jimmy, the Beagley family, and most of all Ellie, whose love, patience, and support means the world to me and has kept me going.

Last but not least, I thank Jesus, my Lord and saviour, for his unfailing love and grace; all to you I owe.

Contents

Chapter 1

Introduction

Programming languages have had, and continue to have, a significant role in advancing computer technology and increasing its utility. By providing layers of *abstraction*, hiding details and allowing reuse, programming languages allow increasingly complex systems to be constructed and managed more effectively.[1] The plethora of existing programming languages (numbering in the thousands) offer varied systems of expression for *organising* and *structuring* computation, helping us to better express ideas and solve increasingly complex problems.

The broad aim of this dissertation is to further advance programming technology, facilitating greater expressivity and hence greater productivity. The dissertation focuses on the programming of *contextual computations*, that is, computations which depend on some notion of the *context* of execution. A class of contextual computations is explored from the perspective of programming and programming languages.

Chapter structure. This introductory chapter explains the general approach and principles of this dissertation, and defines its scope. Section 1.1 briefly introduces the general approaches, methodologies, and themes of the dissertation. Section 1.2 introduces contextual computations and motivates their study. Section 1.3 provides an overview of the dissertation's structure.

1.1. A principled approach to language design

Although most programming languages can in principle encode any computation as a program (*i.e.*, the property of *Turing completeness*), there is no single existing language perfectly suited to expressing succinctly and eloquently every possible computation.[2] Such is the breadth and depth of computational possibilities. Effective programming languages therefore make trade-offs between different aspects of programming, maximising some desirable characteristics of a language at the expense of others.

Design goals for effective languages. The desirable characteristics of a programming language can be broadly summarised by the following four actions on programs which an effective language should benefit: *reading*, *writing*, *running*, and *reasoning* (*the four Rs*) [**Orc11**]. That is, a programming language should improve understandability (reading) and expressivity (writing) for programs, should permit efficient execution (running) of programs, and allow accurate reasoning about their behaviour. These four tenets are not mutually exclusive, but are interrelated. For example, a language in which it is easier to reason about programs, even informally, may be easier to read and understand; or a language where the compiler can easily reason about a program may be more amenable to optimisation.

[1] A good introduction to complexity in software systems is given in the introductory chapter of *Object-oriented analysis and design with applications* by Booch *et al.* [**BME+07**].
[2] Note the difference between *program* and *computation*; a program gives an encoding of a computation.

Improving the four *R*s for a language therefore has non-orthogonal effects, thus maximising all four in one language, for all programs, has proven difficult; often improving one diminishes another. Instead, a language may strongly support the four *R*s for some *classes* of computation, with some accepted disadvantage to others. This inescapable trade-off between characteristics is evidenced by the broad spectrum of programming languages over the last 70 years.[3]

Classifying computations. Given the vast spectrum of possible computations, classifying computations based on common properties is beneficial for programming, where a language, or particular features of a language, may be designed to support particular classes. As Landin argued in his seminal paper, *The Next 700 Programming Languages*, the logical differences and similarities between languages should be identified such that a language can be developed from a "well-mapped" space, systematically derived from logical components, rather than devised "labouriously" from the ground up *ad hoc* [**Lan66**]. By classifying computations the logical components of a language can be more easily designed and motivated.

This dissertation introduces the class of *contextual computations*, exploring the structure of this class and its use, and developing languages and language constructs for its programming.

But how can computations be classified? There are many possible classification criteria. A well known and well studied example is the asymptotic complexity of a computation relative to its input size [**Pap03**], but this is not the approach of this dissertation. Instead, classification here is by the *structure* of the operations and data that comprise a computation. This kind of classification can be seen in the software engineering discourse of *design patterns*, providing reusable schemes, or *programming idioms*, for solving particular programming problems, describing classes of computation with a common form. The discourse of design patterns is particularly prevalent in object-oriented programming (*e.g.*, [**BME+07, Gam95**]). While some of these patterns are specific to the object paradigm, the general approach is valuable in other programming paradigms. In *functional programming*, design patterns are often encapsulated by interfaces of functions and *higher-order* functions, such as in the approach of *algorithmic skeletons* for parallel programming [**Col89, DFH+93**].

Design patterns in functional programming are mainly guided by the use of *type theory*, which provides a framework for analysing and classifying the structure of both computations and data. This approach is applied in this dissertation.

Type theory. Founded in the work of Whitehead and Russell's *Principia Mathematica* [**WR12**], type theory was introduced to rule out paradoxical mathematical constructions. Applied to programming, type theory provides an organising principle for computations and data, aiding program reasoning by excluding large classes of incorrect and inconsistent programs which describe nonsensical and paradoxical computations, hence the slogan: "*well-typed programs do not go wrong*" [**Mil78**]. Barendregt likens types to physical quantities where type checking excludes, for example, the addition of non-comparable quantities, that is, of different *dimension*, *e.g.*,

[3] Of course, the spectrum of languages is not only a consequence of objectively balancing language characteristics, but also the inevitable subjective preferences of people.

"adding 3 volts to 2 ampères" [**Bar93**]. Furthermore, types give an abstract specification of data and the behaviour of a computation which, from the perspective of the *four Rs* for effective languages, aid the reading and writing processes of programming, and the efficient execution of programs, for example in the precise allocation of memory.

A *type system* describes and implements a type theory for a language. One perspective is that type systems are static program analyses which reject programs that violate basic properties of the language (sometimes called language *invariants*). Developments in type theory have allowed this analysis to be more precise, classifying programs not only based on *what* they compute, but also *how* they compute; for example *dependent type theory* allows types to depend on values, elucidating the role of parameters in the computation of a result [**MLS84**]. At the same time, developments in type theory have allowed types to describe the general applicability of a computation to different forms of data; for example, *polymorphic* type theories allow operations defined generically for any type (see, *e.g.*, [**Pie02**, Chapter V]).

These richer type theories, providing *parameterisation* and *indexing* of types, allow more information about a computation to be captured, encoding more nuanced program properties and providing more opportunities for optimisation and enforcing correctness. For example, Barendregt's example of types excluding addition of quantities with different dimension can be made more flexible by indexing a type of real numbers by dimensions, $\mathbb{R}_d$, where addition is typed $+ : \forall d \,.\, \mathbb{R}_d \times \mathbb{R}_d \to \mathbb{R}_d$ but multiplication is typed $\times : \forall d_1, d_2 \,.\, \mathbb{R}_{d_1} \times \mathbb{R}_{d_2} \to \mathbb{R}_{d_1 \otimes d_2}$ where $\otimes$ is a dimension product. This dissertation follows a typed approach to programming, where types with increasing *information content* will be used to describe details of computations.

The structure of programs and data, as quantified by types, can be organised further by principles from abstract mathematics, particularly the field of *category theory*.

Category-theoretic approach. Category theory is a branch of mathematics providing a framework for the abstract description, characterisation, and study of structure in mathematics and formal systems. Most usefully, the abstractions provided by category theory frequently expose common structures and axioms between systems, allowing results to be transferred and reused. For example, the abstract concept of a *(symmetric closed) monoidal category* is a ubiquitous, underlying structure in many concrete topics, such as logic, topology, computation, physics, and language [**BS11**]. This dissertation distinguishes two approaches to applying category theory in programming: *categorical semantics* and *categorical* (or *category-oriented) programming*.

Categorical semantics uses category theory as a metalanguage for defining the semantics of a language, where terms are interpreted in a category that has enough additional structure to satisfy the language properties. For example, Lambek showed the semantics of the simply-typed λ-calculus are captured by *Cartesian-closed* categories [**Lam80**].

Categorical programming on the other hand uses category-theoretic concepts directly in programming as design patterns for organising, structuring, and abstracting (or *modularising*) programs, simplifying both definitions and reasoning. For example, the concept of a *monad* is

used frequently in functional programming to abstract the composition of effect-producing expressions as promoted by Wadler [**Wad92a**, **Wad95b**]. Whilst categorical semantics provides a categorical interpretation to both the type and term structure of a program, categorical programming instead provides a shallow category-theoretic interpretation of just the type structure, providing a framework for structuring programs using category-theoretic concepts.

Categorical programming can be seen as a subset of the *algebra-of-programming* approach.

Calculational/algebraic approach to programming. The algebra-of-programming approach views programs as symbolic equations which can be manipulated and reasoned about similarly to algebraic systems in mathematics [**BDM97**]. The approach of *calculating functional programs* applies the algebraic approach to both derive and reason about programs in a style similar to algebraic calculation [**Gib02**]. Much of the algebraic/calculational programming approach is motivated by, and derived from, concepts in category theory, where total functional programs are placed in correspondence with morphisms of a category [**FOK92**].

Functional languages are particularly suited to this algebraic programming approach because their increased *purity*, controlling and restricting *side effects*, tends to imply stronger algebraic properties with fewer exceptional cases. Additionally, the syntax of functional languages is usually lightweight, permitting easy symbolic manipulation and is usually sufficiently abstract that higher-level computational concepts, rather than implementation details, form the basic building blocks [**Bac78**]. Functional programming is the preferred paradigm of this dissertation.

1.2. Contextual computations

Contextual computations are those that depend upon, or make demands on, the *context* of evaluation.[4] Context is a sufficiently abstract notion that it has many definitions and usages. The following discusses various notions of context which are all related to, or examples of, the class of contextual computations studied in this dissertation. This class of contextual computations is introduced more formally in Section 1.2.2, and related back to these examples.

1.2.1. Notions of context

> *context*, **n**. [1] the circumstances that form the setting for an event, statement, or idea; [2] the parts that come immediately before and after a word or passage and make its meaning clear.
>
> *The Oxford English Dictionary* [**OED06**]

Context in programming language theory. The term *context* appears in two notable situations in programming language theory. Firstly, in type theory, the free-variable typing assumptions of a judgment are called the *context*, where $\Gamma \vdash e : \tau$ relates a term e with a type τ in the context Γ. The name *context* is appropriate since the value of an expression depends on its free-variables, which are provided by the environment.

[4] An alternate name might be *context-dependent* computations, although the shorter *contextual* computation is more convenient and more general in tone – *cf. effect-producing* computations *vs. effectful* computations.

Secondly, *evaluation contexts* for a language are terms with a *hole* that can be *filled* with another term. For example, evaluation contexts of the λ-calculus are described by the grammar:

$$C := [.] \mid \lambda v\,.\,C \mid C\,t \mid t\,C$$

where t ranges over terms without a hole, v ranges over variables, and [.] denotes the *context hole*. Thus, C defines a syntax tree with a single context hole that may be filled (or plugged) by another term, written $C[t]$. Using evaluation contexts, a common notion of semantic equivalence between expressions is *contextual equivalence*, in which two expressions are considered equal (for some definition of equality) if their substitution into any context preserves this equality: [**Pie05**, p.249] (*i.e.*, $\equiv$ is a *congruence relation*) where $\forall C.\, t \equiv t' \;\Rightarrow\; C[t] \equiv C[t']$.

Context of execution. From a more applied perspective, the field of *context-aware computing*, a subset of *pervasive computing* research, concerns building applications that adapt "according to the location of use, the collection of nearby people, hosts, and accessible devices, as well as to changes to things over time" [**SAW94**]. In this sense, *context* encapsulates (possibly hidden) parameters to a computation from the outside world, which are out of a program's control. Context-aware computing has developed formal programming constructs, birthing the field of *context-oriented programming* for which context is broadly defined as "everything that is computationally accessible" [**HCH08**]. Computations may be executed on different devices with different resources (phone, server, the "cloud", *etc.*) which each provide a notion of context to a computation. Various programming languages support different execution environments directly as language primitives, capturing context-dependence properties [**MCH08**, **CLWY06**, **SC10**].

Context in language. In natural language, the role of context in the meaning of sentences has long been recognised and studied for centuries by logicians and philosophers. Notably, Frege's influential 1892 paper *Über Sinn und Bedeutung* (On Sense and Meaning), discussed problems of equality between terms that have the same *reference* (*denotation* à la Russell) but different *sense*, or *meaning*, for which *context* affects the validity of substitution for apparently equal terms [**Fre48**] (1948 English translation). Frege's classic example considers the two expressions "the morning star" and "the evening star", both referring to the planet Venus, but with different meanings (sense): Venus as seen at sunrise and sunset respectively.

The difference between reference and sense is exposed by substituting *equi-designating* terms into a sentence which has some implicit *indirect reference context* [**Fit09**]. For example, the truth of the statement "John knows that the morning star is seen at sunrise" may differ to "John knows that Venus is seen at sunrise" because of the implicit context of John's knowledge.

Carnap introduced the terms *intension*, describing the entire sense/meaning of a term, and *extension* describing just the denotation or object to which an expression refers, describing intensions mathematically as functions from states (contexts) to extensions [**Car47**]. Carnap's ideas created the field of *intensional logic* for reasoning about contextual and modal systems.

Context and dataflow. Intensional logic was influential in the development of the dataflow programming language Lucid, intended to make iterative, imperative programming more declarative. Following its inception, it was realised that Lucid provides an intensional view of imperative, iterative programming, denoting expressions as *streams* mapping from a discrete, abstract notion of time to extensional values [**WA85**]. Thus, a variable in an imperative language, which may have many denotations during a program, is captured in Lucid as an intensional expression capturing the entire history of its values during the course of computation. *Intensional operators* provide access between contexts. Later, Lucid was extended to multi-dimensional contexts allowing intensional programs over array-like structures [**AFJW95**].

Context in local/global computation. Many programs manipulate structured data, such as lists, arrays, databases, stacks, trees, and graphs. Many such manipulations are characterised by *aggregate*, or *bulk*, operations on a data structure that can be separated into two parts: a *local computation*, defined at a particular point or *context* within the data structure, and a scheme for *promotion* (*extension*) of the computation to a *global computation*, *i.e.*, to all contexts.

As an example of a local computation, the *discrete Gaussian* operator, used in signal and image processing, can be calculated at some position i of a one-dimensional array A as follows:

$$\tfrac{1}{4}(A[i-1] + 2A[i] + A[i+1]) \tag{1}$$

The discrete Gaussian of an entire array can be computed by iterating over its index space, calculating equation (1) for each index (*i.e.*, globally) and collecting the results into an array of the same size. For example, in some imperative language the promotion operation may be written (for this discussion, ignoring issues of *boundary conditions*):

$$\textbf{for } i \in \mathit{dom}(A) : \; B[i] = (\mathit{eq.}\ 1) \tag{2}$$

This iterative *promotion* operation is standard and can be reused for other operations, for example, with a local mean of a neighbourhood of five elements, defined:

$$\tfrac{1}{5}(A[i-2] + A[i-1] + A[i] + A[i+1] + A[i+2]) \tag{3}$$

Another example of a local operation is the *live variable analysis* (LVA) defined over the nodes of a *control-flow graph* (CFG) [**App97**]. The LVA calculates information for each node from the current node and its successors, as described by the local operation on a CFG G, at node n:

$$\bigcup_{s \in \mathit{succ}(G,n)} (\textsc{Gen}[s] \cup (\textsc{Live}[s] - \textsc{Kill}[s])) \tag{4}$$

where $\mathit{succ}(G, n)$ computes the set of successors for the node n in graph G. This local operation is promoted globally to the entire CFG by the iterative code:

$$\textbf{for } n \in [\mathit{final}\ G..\mathit{start}\ G]\ \{\textsc{Live}'[n] = (\mathit{eq.}\ 4)\}$$

which computes one pass of the live-variable analysis (usually the global operation is applied iteratively until a fixed-point is reached).

The local operations above access data elements relative to, or *neighbouring*, the *current* element, computing a result value for this context. Local operations depending on their neighbours are more general than local operations depending on just the current element (*cf.*, the *map* operations on lists). This local-operation pattern is ubiquitous in programming, particularly in scientific modelling; *e.g.*, Smith's dissertation on biological and geometric modelling is motivated by the philosophy of *local modelling* over inherently global models [**Smi06**].

1.2.2. Structure

There are likely other notions of context not discussed here. Those included are particularly relevant to this dissertation, although any omitted may also be relevant. The last example of context, of local operations with a global-promotion scheme, was given a more concrete treatment as it has a straightforward mapping to the category-theoretic structure of a *comonad*, which forms the basis for the class of contextual computations in this dissertation.

The example of a local operation for an array, can be described generally by a function of type $T\,A \times P \to B$, for some parametric data type/data structure T with element type A, return type B, and a type P of *positions* (alternatively, *indices*), at which the local operation is applied, where P has the pre-condition that its values are within the domain of positions for the parameter $T\,A$. For the graph example, the notion of a position was less explicit, but may be an index-like value as with arrays or some other *structural* notion of position.

Both kinds of local operation are captured by a function of type $CA \to B$ where the parametric type CA encodes the notion of position in some way. The category theory structure of a comonad over C provides a global promotion operation for these local operations and a trivial local operation which defines the notion of the *current context*.

Comonads define a subset of the class of contextual computations studied in this dissertation.

Definition 1.2.1. Informally, from a functional programming perspective, a *comonad* comprises a parametric data type C with the following two polymorphic operations, which satisfy a number of laws (shown later):

$$\mathit{current} : \forall \tau \,.\, C\tau \to \tau$$

$$\mathit{extend} : \forall \sigma, \tau \,.\, (C\sigma \to \tau) \to C\sigma \to C\tau$$

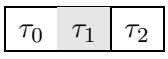
Values of type $C\tau$ represent contextual values of type τ, and will be called *intensions*, in the sense of Carnap [**Car47**] and Montague [**Fit09**], capturing the extensional values of a computation, of type τ, in all possible contexts. Intensions $C\tau$ include an encoding of the *current context*, where the *current* operation returns the extensional value at this context.

Local contextual operations of type $C\sigma \to \tau$ compute an extensional τ value at the context provided by $C\sigma$. The *extend* operation lifts a local contextual operation, of type $C\sigma \to \tau$, applying it to *all contexts* encoded by an intension $C\sigma$, constructing an intension $C\tau$ where an extensional value of type τ at a particular context is computed by the local operation applied at that context. As shall be seen in Chapter 3, *extend* also defines *accessibility* between contexts.

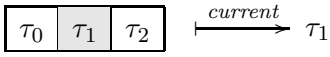
The types of these operations and the comonad laws indicate that *extend* applies the local operation at every context, computing a new local value for each context, possibly from values in neighbouring contexts. Notably, contextual computations have only input dependencies, *i.e.*, there its no output dependency.

Example 1.2.2. As an informal example, arrays describe values in a spatial context whose values are dependent on position (*e.g.*, temperature within a fluid, population at a particular grid reference). Thus for a parametric data structure Array (not a "classical" array):

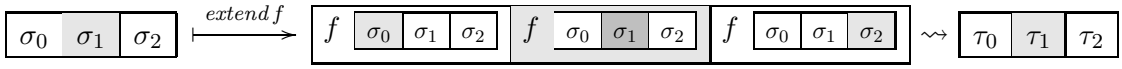
– Array τ encodes contextual computations of τ-values dependent on positions, and includes the current context of the computation, called the *cursor*.

e.g.  $\in$ Array τ has three contextual positions with the current context in grey.

– *current* : Array $\tau \to \tau$ defines the current context as the array element pointed to by the cursor, returning the cursor element.

e.g. 

– *extend* lifts a local operation f : Array $\sigma \to \tau$ to a global by applying f at each context, where *extend* f : Array $\sigma \to$ Array τ:

e.g. 

Comonads capture a class of contextual computations which will be studied in this dissertation. However, this dissertation considers a broader notion of contextual computations, including those structured by comonads, captured by the notion of an *indexed comonad*. Informally an indexed comonad is defined:

Definition 1.2.3. An *indexed comonad* comprises a *family* of types C_R indexed, or *tagged*, by information R about the encoded context, with operations:

$$\mathit{current} : \forall \tau \,.\, C_{\mathsf{0}} \tau \to \tau$$
$$\mathit{extend} : \forall \sigma, \tau, R, S \,.\, (C_S \sigma \to \tau) \to C_{R \oplus S} \sigma \to C_R \tau$$

where $\mathsf{0}$ is a particular index denoting *trivial* context-dependence, and $\oplus$ combines contextual information (*cf.*, the dimension-indexed types of $+$ and $\times$ mentioned in Section 1.1).

The indices of an indexed comonad are used to provide more information about contextual computations and their *contextual requirements*, which will be used for improving reasoning and optimisation further than what is possible with the (un-indexed) comonad structure.

The various notions of context described in Section 1.2.1 are all captured by comonads or indexed comonads. For the notion of context provided by data structures with local operations, there is a clear mapping between these operations and this structure. For the more implicit notions of context, such as in *context-aware programming* and *context in natural languages*, a

suitable indexed data type C_R encodes the appropriate notion of context, providing a semantics to the implicit notions of context in programming systems and natural languages.

Why contextual computations? As described above, notions of context in computing are ubiquitous. Nanevski *et al.* point out a number of applications where the role of context is important including "data abstraction, run-time code generation, distributed computation, partial evaluation, and meta-programming" [**NPP08**]. As discussed above, and later in Chapter 3, many data manipulation tasks can be treated as contextual computations. These data transformations are inherently data parallel. Contextual computations are therefore highly relevant given current trends towards parallel and distributed architectures in response to diminishing gains from previous architectural approaches.

1.3. Dissertation outline

- **Chapter 2. Background** – Reviews existing relevant background material on the simply-typed λ-calculus as a foundation for programming and the approaches of *categorical semantics* and *categorical programming*. The presentation provides a schema for categorical semantics of the simply-typed λ-calculus, which is applied in Chapter 3 and 6 for contextual calculi.
- **Chapter 3. Comonads** – Reviews comonads in programming and semantics, introducing new examples, and contributing a semantics for the *contextual λ-calculus* that builds on the work of Uustalu and Vene [**UV08**]. Comonads are shown to have a *shape preservation* property, which restricts their utility, which is relaxed in Chapter 5 to generalise comonads.
- **Chapter 4. A notation for comonads** – Contributes a lightweight notation for categorical programming with comonads, facilitating contextual programming embedded in Haskell.
- **Chapter 5. Generalising comonads** – Introduces various generalisations of comonads, most notably the novel structure of an *indexed comonad* which widens the class of contextual computations by relaxing shape preservation.
- **Chapter 6. The coeffect calculus** – Introduces a general class of static analyses for context-dependence in programs called *coeffect systems*. Indexed comonads are used to embed this analysis in a generalised context-dependent calculus, called the *coeffect calculus*, which generalises the contextual λ-calculus of Chapter 3.
- **Chapter 7. A contextual language for containers** – Draws together the work of the preceding chapters into a single language for efficient and correct contextual programming with containers (mainly arrays) embedded in Haskell. The language is parameterised by indexed comonads with a coeffect system which provides safety guarantees and enables optimisations.
- **Chapter 8. Conclusions and further work** - Concludes and discusses general further work with some initial thoughts.

Each chapter concludes with relevant further and related work where appropriate.

Chapter 2

Background

This dissertation uses category theory to structure both language semantics (categorical semantics) and programs (categorical programming). This chapter reviews relevant background material from the literature and is largely pedagogical. Knowledge of categories, functors, and natural transformations is assumed. An introduction to these concepts is given in Appendix A (p. 181). Further relevant categorical notions are introduced as needed, primarily from the perspective of categorical semantics for the λ-calculus.

The λ-calculus as a foundation. The kind of programming considered in this dissertation is that of *functional programming* for which Church's λ-calculus is the prototypical, or foundational, functional language. *Typed* variants of the λ-calculus define a subset of valid terms via classification by type, enforcing logical consistency by excluding irreducible, nonsensical, and paradoxical computations. The *simply typed* and *polymorphic λ-calculi* form the basis of modern day functional languages such as the ML family of languages, Haskell, F#, and OCaml.

The credibility of the λ-calculus as a basis for programming is supported by the correspondence between the type-theory of the simply-typed λ-calculus (hereafter abbreviated λ^τ) and intuitionistic propositional logic. This is known as the *Curry-Howard correspondence* (or more strongly: *isomorphism*). *Propositions* and *proofs* of intuitionistic logic, in natural deduction form, are *homologous*, *i.e.*, have the same structure and shape, to the *types* and *terms* of λ^τ. A well-typed term corresponds to a proof of the type's corresponding proposition, thus proving formulae and writing programs are equivalent activities [**GTL89**]. Intuitionistic implication, and its introduction and elimination rules, have a one-to-one correspondence with the function-space type and the rules for abstraction and application (function construction and deconstruction). *Product* (tuple) and *coproduct* (sum/union) types can be added to λ^τ corresponding to conjunction and disjunction in propositional logic.

As an analytical tool, intuitionistic logic allows reasoning under *hypothetical evidence*. Hypotheses in intuitionistic proofs correspond to *free variables* in λ-terms, where *cut elimination* (discharging hypotheses) corresponds to *substitution* for λ^τ (eliminating free variables):

$$[\text{cut}]\,\frac{\Gamma, B \vdash A \quad \Gamma' \vdash B}{\Gamma, \Gamma' \vdash A} \quad \cong \quad [\text{subst}]\,\frac{\Gamma, x : \tau' \vdash e : \tau \quad \Gamma' \vdash e' : \tau'}{\Gamma, \Gamma' \vdash e[x := e'] : \tau} \tag{5}$$

Intuitionistic propositional logic and the simply-typed λ-calculus thus have equivalent *analytical power* with a common notion of *abstraction* and *substitution*.

Category theory as a foundation. Lambek showed that both the intuitionistic propositional logic and the type theory of the simply-typed λ-calculus are modelled by *Cartesian-closed categories* [**Lam80**, **LS88**], elucidating the Curry-Howard correspondence formally by capturing their common structure. This correspondence shows that Cartesian-closed categories (hereafter

CCCs) have enough extra structure on top of the basic category structure to axiomatise the notion of substitution common to both intuitionistic logic and λ^τ.

The mapping of type theory and logic to a common categorical structure is called the *extended* Curry-Howard correspondence. In this extended correspondence, *objects* of a CCC correspond to types and propositions, and *morphisms* to terms and proofs, mapping from a context of free variables (hypotheses) to results (conclusions). In Lambek's approach, categorical *products* provide concatenation of hypotheses/free variables (discussed in Section 2.2). Substitution in λ^τ and hypothesis discharge in intuitionistic logic correspond to *composition* in a CCC:

$$(eq.\,5) \;\cong\; \frac{\Gamma' \xrightarrow{g} B \qquad \Gamma \times B \xrightarrow{f} A}{\Gamma \times \Gamma' \xrightarrow{\langle id, g\rangle} \Gamma \times B \xrightarrow{f} A} \tag{6}$$

The approach to modelling logic and language using category theory is known as *categorical logic*, where the language component provides a *categorical semantics*. Since simply-typed λ-terms correspond to CCC morphisms, terms of a CCC, comprising compositions of morphisms (and functors, natural transformations), can be replaced by equivalent, but more syntactically compact, λ-terms. Thus, λ^τ is described as an *internal language* of CCCs. The logical correspondents of categorical models and type theories will not be considered in this dissertation.

Various category-theoretic notions can be used to model different notions of computation. Relevant to this dissertation is Moggi's use of *strong monads* for a semantics of side-effects, such as non-determinism, state, and exceptions [**Mog89, Mog91**]. Moggi's *computational* λ*-calculus* (having a weaker equational theory to the simply-typed λ-calculus) captures various notions of effectful computation. This approach is dualised by Uustalu and Vene's categorical semantics of a context-dependent λ-calculus in terms of *monoidal comonads* [**UV08**], discussed in Chapter 3.

Chapter structure. In this dissertation, typed-languages are introduced with an equational theory, defining what it means for two terms to be equal, and a supporting categorical semantics. Section 2.1 and Section 2.2 exemplify this approach for the simply-typed λ-calculus, which is taken as a foundation for the rest of the dissertation. Section 2.1 introduces the syntax, typing, and equational theory of the simply-typed λ-calculus, as well as fixing notation. Section 2.2 describes a categorical semantics in terms of CCCs along the lines of the Lambek approach.

Section 2.3 briefly introduces Moggi's *computational* λ-calculus for effectful computation, captured by strong monads. Section 2.4 reviews the approach of *categorical programming*, in which categorical concepts are used directly in programming providing idioms for organising and abstracting programs. Programming examples in this dissertation are in Haskell [**PJ+03**].

Notation review. Category theory and type theory are full of *arrows*. Throughout, the following arrow symbols are used most commonly for the following tasks:

- $\to$ – morphisms and Haskell function types;
- $\Rightarrow$ – exponential objects, λ-calculus function types, and Haskell constraints, *e.g.*, $C \Rightarrow \tau$ is type τ with constraint C;

- $\dot{\to}$ – natural transformations, *e.g.*, $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ (sometimes written instead with parameters as a morphism $\alpha_X : \mathsf{F}X \to \mathsf{G}X$);
- $\rightsquigarrow$ – reductions in an operational semantics;

Proofs proceed in a *direct*, *calculational*-style, by applying axioms/equalities to reduce equations, and are written in the following form:

$$\begin{array}{ll} & E \circ F \circ G \\ = & E \circ F' \circ G \quad \{\textit{property providing } F = F'\} \end{array}$$

For a category $\mathbb{C}$, $\mathbb{C}_0$ denotes its class of objects, $\mathbb{C}_1$ its class of morphisms, and $\mathbb{C}(A, B)$ the *hom-set* (*i.e.*, collection of morphisms) between $A, B \in \mathbb{C}_0$.

2.1. The simply-typed λ-calculus

The presentation here discusses only the pertinent details. Many textbooks give a more thorough account, *e.g.*, that of Hindley and Seldin [**HS08**] which was used as a reference for this section.

2.1.1. Syntax and typing

The simply-typed λ-calculus comprises two syntactic classes of *expressions* (or *terms*) and *types*:

$$e ::= \lambda v \,.\, e_1 \mid e_1\, e_2 \mid v \qquad\qquad \tau ::= \tau_1 \Rightarrow \tau_2 \mid T$$

where v ranges over variables and T over *base types* (type constants). Throughout, σ and τ range over types. A common presentation includes syntactic *type signatures* on binders of λ-terms to assist type inference. This is elided here for simplicity since inference is not a primary concern.

Traditionally, a *typing* relation between terms and types is denoted by a single colon, $(:) \subseteq (e \times \tau)$. However, this relation types only *closed terms* (without free variables). Both closed and open terms are typed by *typing judgments* $\Gamma \vdash e : \tau$ with a sequence Γ of *typing assumptions* for free variables. Terms of λ^τ have the following typing rules:

$$[\textsc{abs}]\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \lambda x \,.\, e : \sigma \Rightarrow \tau} \quad [\textsc{app}]\frac{\Gamma \vdash e_1 : \sigma \Rightarrow \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\, e_2 : \tau} \quad [\textsc{var}]\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \tag{7}$$

Structural typing. The following *structural rules* involve types in the context of a judgment rather than terms, making explicit various implicit meta-rules of typing:

$$[\textsc{weaken}]\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \quad [\textsc{exchange}]\frac{\Gamma, x : \sigma, y : \tau, \Gamma' \vdash e : \tau}{\Gamma, y : \tau, x : \sigma, \Gamma' \vdash e : \tau} \quad [\textsc{contract}]\frac{\Gamma, y : \tau', z : \tau' \vdash e : \tau}{\Gamma, x : \tau' \vdash e[z := x][y := x] : \tau} \tag{8}$$

Thus, redundant typing assumptions can be added (*weakening*), assumptions can be permuted (*exchange*), and two free variables of the same type may be replaced by one (*contraction*).

Each of these structural rules is *admissible*, *i.e.*, adding the rule does not increase the number of terms that can be typed, or equivalently, a term whose type derivation uses the admissible rule can be equally typed by a derivation without the rule; admissible rules are redundant. For example, [APP] admits contraction-like behaviour where Γ is "duplicated" and passed to

its subterms (the categorical semantics for application, shown later, makes this duplication of context explicit). The [VAR] rule includes implicit exchange and weakening.

The standard typing rules (7) can be reformulated such that weakening, contraction, and exchange are no longer admissible, with alternate rules for application and variable access:

$$[\text{APP}]\frac{\Gamma \vdash e_1 : \sigma \Rightarrow \tau \qquad \Gamma' \vdash e_2 : \sigma}{\Gamma, \Gamma' \vdash e_1\, e_2 : \tau} \qquad [\text{VAR}]\frac{}{x : \tau \vdash x : \tau} \tag{9}$$

Sub-structural systems omit some or all of the structural rules. Section 6.6.2 describes a calculus with rules of this form. However, this dissertation mostly uses the typing rules of (7).

Substitution. Essential to the reduction and equational theory of λ-terms is the meta-theoretic notion of *substitution*, where a term of appropriate type can be substituted for a free variable by the following syntactic rewrite (where $FV(e)$ computes the set of free variables for a term e) (and assuming α-equivalence classes of λ-calculus terms, see Section 2.1.2 below):

$$\begin{array}{lcll} (v)[x := e] & \equiv & v & \text{iff } x \neq v \\ (x)[x := e] & \equiv & e & \\ (\lambda v.e_1)[x := e] & \equiv & \lambda v.(e_1[x := e]) & \text{iff } x \neq v, \text{ where } v \notin FV(e) \\ (e_1\, e_2)[x := e] & \equiv & e_1[x := e]\, e_2[x := e] & \end{array} \tag{10}$$

Lemma 2.1.1 (Substitution). *Given expressions e and e' with type derivations, $\Gamma, x : \tau' \vdash e : \tau$ and $\Gamma \vdash e' : \tau'$, then $\Gamma \vdash e[x := e'] : \tau$.*

Proof. Straightforward, by rule induction over $\vdash$ (structural induction over inference rules). □

The action of the substitution lemma (2.1.1) is sometimes *internalised* (*i.e.*, imported to the syntactic-level from the meta-level) as the *let-binding* construction, with syntax and typing:

$$e ::= \ldots \mid \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 \qquad\qquad [\text{let}]\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, v : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \tau}$$

2.1.2. Equational theory

Semantically equal terms of the same type τ, in a context Γ, are described by the relation $\equiv$ written $\Gamma \vdash e \equiv e' : \tau$. This relation is a *congruence* with the usual axioms of an *equivalence relation* (reflexivity, symmetry, transitivity) and congruence axioms over the structure of terms:

$$[\equiv\text{-app-cong}]\frac{\Gamma \vdash e_1 \equiv e_1' : \sigma \Rightarrow \tau \qquad \Gamma \vdash e_2 \equiv e_2' : \sigma}{\Gamma \vdash e_1\, e_2 \equiv e_1'\, e_2' : \tau} \qquad [\equiv\text{-}\lambda\text{-cong}]\frac{\Gamma, x : \sigma \vdash e \equiv e' : \tau}{\Gamma \vdash \lambda x.e \equiv \lambda x.e' : \sigma \Rightarrow \tau}$$

An important axiom of the theory, known as α-equivalence, is that bound variables can be arbitrarily renamed (if the renaming does not overlap with another variable), *i.e.*:

$$[\equiv\text{-}\alpha]\frac{}{\Gamma \vdash \lambda x.e \equiv \lambda y.e[x := y] : \sigma \Rightarrow \tau}\, y \notin FV(e)$$

Local soundness and completeness. Pfenning and Davies introduced the terminology of *local soundness* for a logical connective ϕ if a proof P containing occurrences of the introduction rule for ϕ followed immediately by its elimination rule can be simplified to an equivalent proof P' by

removing the introduction-elimination pair [**PD01**], *i.e.*, in natural deduction style:

$$\begin{array}{l}[\phi\text{-I}]\\ [\phi\text{-E}]\end{array}\frac{\Gamma \vdash A}{\Gamma \vdash \phi\, A} \quad \equiv \quad \Gamma \vdash A \qquad \text{(local soundness of } \phi\text{)}$$

Local soundness implies that, in Pfenning and Davies words, "elimination rules are not too strong" [**PD01**]; that is, the conclusion can be constructed from just the premises.

The dual of local soundness is *local completeness*: that eliminating a connective ϕ provides enough evidence such that the original formula can be reconstructed:

$$\Gamma \vdash \phi A \quad \equiv \quad \begin{array}{l}[\phi\text{-E}]\\ [\phi\text{-I}]\end{array}\frac{\Gamma \vdash \phi A}{\Gamma \vdash A}\qquad \text{(local completeness of } \phi\text{)}$$

Thus elimination of a connective ϕ followed immediately by an introduction of ϕ can be elided.

Prawitz' seminal monograph on natural deduction develops the same notion (his *inversion principle*) where "a consequence of an I-rule which is also a major premiss of an E-rule constitutes a complication in a deduction. As such a complication can be removed." [**Pra65**, pp. 32-38]

Via the Curry-Howard correspondence, terms constructing and deconstructing a type constructor correspond to introduction and elimination rules. Local soundness and completeness correspond to properties known as β- and η-*equivalence* respectively, shown in **Figure** 2.1, where Φ and Φ^{-1} construct and deconstruct values of the ϕ type.

The function type $\Rightarrow$ of λ^τ corresponds to intuitionistic implication with introduction by abstraction and elimination by application, which are both locally sound and complete by the equalities on type derivations in **Figure** 2.2. Thus $\beta\eta$-equality holds for function-typed terms:

$$[\equiv\text{-}\beta]\frac{\Gamma, x : \sigma \vdash e : \tau \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash (\lambda x.e)\, e' \equiv e[x := e'] : \tau} \qquad [\equiv\text{-}\eta]\frac{\Gamma \vdash e : \sigma \Rightarrow \tau}{\Gamma \vdash (\lambda x.e\ x) \equiv e : \sigma \Rightarrow \tau}\ x \notin FV e$$

Note that **Figure** 2.2 shows only the syntax-level equivalence of the terms and not the semantic-level equivalence, which is considered for the categorical semantics of λ^τ in Section 2.2.

If *let*-binding syntax has been included into the calculus to internalise syntactic substitution then an additional, but equivalent, local soundness rule can be provided for $\Rightarrow$:

$$[\equiv\text{-}let\text{-}\lambda]\frac{\Gamma, x : \sigma \vdash e : \tau \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash (\lambda x.e)\, e' \equiv \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau}$$

This rule is sometimes taken as an axiom (indeed, Landin first introduced *let* as syntactic sugar for abstraction followed by application [**Lan66**]). By transitivity, $[\equiv\text{-}\beta]$, and $[\equiv\text{-}let\text{-}\lambda]$, then:

$$[\equiv\text{-}let\text{-}\beta]\frac{\Gamma, x : \sigma \vdash e : \tau \qquad \Gamma \vdash e' : \sigma}{\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e \equiv e[x := e'] : \tau}$$

$$\begin{array}{l}[\phi\text{-I}]\\ [\phi\text{-E}]\end{array}\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Phi\, e : \phi\, \tau} \qquad [\equiv\text{-}\beta]\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Phi^{-1}\Phi\, e \ \equiv\ e : \tau} \qquad [\equiv\text{-}\eta]\frac{\Gamma \vdash e : \phi\, \tau}{\Gamma \vdash \Phi\Phi^{-1}\, e \ \equiv\ e : \phi\, \tau}$$

(a) Local soundness (β-equivalence) (b) Local completeness (η-equivalence)

Figure 2.1. Curry-Howard correspondents of local soundness and completeness.

$$\begin{array}{l}[\text{ABS}]\\[\text{APP}]\end{array}\frac{\dfrac{\Gamma, x:\sigma \vdash e:\tau}{\Gamma \vdash \lambda x\,.\,e:\sigma\Rightarrow\tau} \qquad \Gamma \vdash e':\sigma}{\Gamma \vdash (\lambda x\,.\,e)\,e':\tau} \quad\equiv\quad \Gamma \vdash e[x := e']:\tau$$

(a) Local soundness ($\equiv$-β) for $\Rightarrow$

$$\begin{array}{l}[\text{WEAKEN}]\\[\text{APP}]\end{array}\frac{\dfrac{\Gamma \vdash e:\sigma\Rightarrow\tau}{\Gamma, x:\sigma \vdash e:\sigma\Rightarrow\tau} \qquad [\text{VAR}]\,\dfrac{x:\sigma \in (\Gamma, x:\sigma)}{\Gamma, x:\sigma \vdash x:\sigma}}{[\text{ABS}]\,\dfrac{\Gamma, x:\sigma \vdash e\,x:\tau}{\Gamma \vdash \lambda x\,.\,e\,x:\sigma\Rightarrow\tau}} \quad\equiv\quad \Gamma \vdash e:\sigma\Rightarrow\tau$$

(b) Local completeness ($\equiv$-η) for $\Rightarrow$

Figure 2.2. Equalities between derivations for local soundness and completeness.

For calculi other than the pure simply-typed λ-calculus in this dissertation, this equivalence may not hold as *let*-binding may not be a straightforward internalisation of syntactic substitution.

Extensionality. For functions, η-equality corresponds to *extensional equality*: two functions are equal if they have the same output for the same input, for all possible inputs, *i.e.*:

$$[\text{ext}]\frac{\Gamma, x:\sigma \vdash e\;x \equiv e'\;x:\tau}{\Gamma \vdash e \equiv e':\sigma\Rightarrow\tau} \quad x \notin FV(e), x \notin FV(e')$$

The [ext] rule could be generalised to arbitrary terms $\forall t\ \Gamma \vdash t:\sigma$ with $\Gamma \vdash e\ t \equiv e'\ t:\tau$ in its premise. This general form is equivalent to [ext] but requires more detailed proofs with infinite trees. Appendix C.1.1 (p. 195) demonstrates the admissibility of [ext] in the presence of [$\equiv$-η] for functions, and vice versa, by defining each in terms of the other.

Various other laws hold, for example nested *let*-binding is associative under certain restrictions. Such laws can be derived from the $\beta\eta$-equalities (and [$\equiv$-*let*-λ]) and the properties of syntactic substitution. These properties are made explicit later in Section 2.2.2 (p. 27).

2.1.3. Reduction

Terms of the λ^τ can be evaluated by a *reduction* (simplification) relation $(\rightsquigarrow) \subseteq (e \times e)$, providing a small-step structural operational semantics. The notions of β- and η-equivalence on proof trees imply a possible reduction strategy on terms, where an introduction-elimination pair, or elimination-introduction pair, can be removed, called β- and η-reduction respectively:

$$\Phi^{-1}\Phi\,e \rightsquigarrow_\beta e \qquad\qquad \Phi\Phi^{-1}\,e \rightsquigarrow_\eta e$$

For functions, β- and η-reduction are therefore:

$$\begin{array}{ll}(\beta) & (\lambda x.e_1)\,e_2 \rightsquigarrow e_1[x := e_2]\\ (\eta) & (\lambda x.f\;x) \rightsquigarrow f\end{array}$$

The property of *type preservation* for reductions then follows from $\beta\eta$-equality:

Lemma 2.1.2 (Type preservation)**.** *For all* Γ, e, e', τ *where* $\Gamma \vdash e:\tau$ *and* $e \rightsquigarrow e'$*, then* $\Gamma \vdash e':\tau$*.*

Reduction schemes, such as *call-by-value* and *call-by-name*, describe additional reductions for the λ-calculus. For example, *call-by-name* has an additional rule:

$$(\zeta_1)\frac{e_1 \rightsquigarrow e_1'}{e_1\, e_2 \rightsquigarrow e_1'\, e_2}$$

Such strategies are not considered at length in this dissertation; categorical semantics is the focus rather than operational semantics. Appendix B.1.1 (p. 185) describes the *call-by-value* and *call-by-name* reduction strategies (and *full-beta* for completeness), which are occasionally discussed.

2.1.4. Higher-order type theories

Reasoning under hypotheses is a powerful analytical tool, which the simply-typed λ-calculus describes succinctly at the level of terms by *abstraction*. The *polymorphically-typed* λ-calculus extends the type theory of the simply-typed λ-calculus to include abstraction over types, introducing *type variables*.

Various other type theories provide different kinds of type-level abstraction. For example, *dependent type theory* extends the Curry-Howard correspondence to predicate logic with universal and existential quantification, allowing types to depend upon values [**MLS84**]. This allows further information about a computation to be encoded, widening the scope of possibilities for enforcing program correctness via types.

The languages and semantics defined in this dissertation are for first-order type theories, akin to the simply-typed λ-calculus. However, programming examples are in Haskell, which has *parametric* and *ad-hoc polymorphic* types, and lightweight dependent-types (used in Chapter 7).

2.2. Categorical semantics of the simply-typed λ-calculus

Lambek showed that *Cartesian-closed categories* (CCCs) provide enough additional structure on top of a category to model the simply-typed λ-calculus [**Lam80, LS88**], or equivalently, that the simply-typed λ-calculus is an adequate *internal language* for CCCs. The prototypical CCC is **Set**, the category of sets and functions. This section introduces the approach.

Monoids and notions of composition.
Category theory emerged from the long-standing tradition of *abstract algebra*. Abstract algebra provides a hierarchy of abstract structures for mathematics, building on the simple concept of a total binary operation over a set (sometimes called a *magma*) by adding axioms and operations. A simple, ubiquitous structure in this hierarchy is the *monoid*.

Definition 2.2.1. A *monoid* $(X, \bullet, I)$ comprises a set (or class) of objects X, a (total) binary operation $\bullet$ on X objects, and an element $I \in X$ called the *unit* or *identity* object, such that:

[M1]	(*right unit*)	$x \bullet I = x$	$(\forall x \in X)$
[M2]	(*left unit*)	$I \bullet x = x$	$(\forall x \in X)$
[M3]	(*associativity*)	$x \bullet (y \bullet z) = (x \bullet y) \bullet z$	$(\forall x, y, z \in X)$

Examples 2.2.2 A number of common examples include:

- $(\mathbb{Z}, +, 0)$, $(\mathbb{Z}, \times, 1)$, $(\mathbb{Z}, gcd, 0)$, $(\mathbb{Z}, lcm, 1)$, $(\mathbb{N} \cup \{\omega\}, min, \omega)$, $(\mathbb{N}, max, 0)$, ...
- $(\mathbb{B}, \wedge, \mathbf{T})$, $(\mathbb{B}, \vee, \mathbf{F})$, $(\mathbf{Set}, \cup, \emptyset)$, $(\mathbf{Set}, \cap, \mathbf{S})$ (where $\mathbf{S}$ is the universal set)
- $([a], ++, [])$ $(\forall a)$ where $[a]$ means lists of a elements, $++$ is list concatenation, and $[]$ is the empty list, which is the *free monoid*, *i.e.*, no additional axioms hold.

Monoids also underpin many category theory concepts, including categories, products, monads, monoidal functors, and monoidal natural transformations. Monoids appear also in many other situations. For example, programming languages often exhibit monoidality in program construction. Consider the following equivalences between simple programs in some imperative language, where `c1;c2` means first evaluate the left-hand command then evaluate the right-hand command, `nop` is a unit command, and bracketing groups pairs of composed programs:

```
a = 3;          (a = 3;
(b = 4;    ≡     b = 4;)
 c = 5;)        c = 5;

a = 3;          a = 3;          nop;          a = 3;
nop;       ≡                    a = 3;   ≡
```

Such equivalences seem obvious and natural. Here (`commands`, `;`, `nop`) is a monoid, where the composition of programs `;` is the binary operation of the monoid.

The categorical semantics approach uses categories to model languages, providing a compositional semantics. Categories generalise monoids, where composition is a partial binary operation (not all programs can compose). Equational laws for the λ-calculus are introduced later which correspond to the axioms of a category and resemble the above equivalences.

Categorical semantics is sometimes considered to be a denotational semantics since categorical notions are assigned as meanings for terms in a compositional style, *i.e.*, the denotation of a term is constructed from the denotations of subterms.

General categorical semantics approach. A categorical semantics for a typed language defines a mapping $[\![-]\!]$ from types to objects of some category $\mathbb{C}$ *i.e.* $[\![\tau]\!] \in \mathbb{C}_0$, and from terms to morphisms of $\mathbb{C}$, where the morphisms map between the type of inputs to a term (the free variables) to the type of the output (the result):

$$[\![\overline{v : \tau} \vdash e : \tau]\!] : [\![\overline{\tau}]\!] \to [\![\tau]\!] \in \mathbb{C}_1$$

Note the context is represented here by a vector of variable-type pairs $\overline{v : \tau}$; the interpreted vector of types $\overline{\tau}$ must have some suitable categorical interpretation. Categorical semantics shown in this chapter are *type-directed* rather than *syntax-directed*, defined over typing derivations so that structural rules, which have no associated syntax, are included in the semantics.

2.2.1. Single-variable context and *let*-binding: categories

A category (Definition (A.1.1), p. 181) with no additional structure models a simple language, call it L_0, that has a binding syntax (*let*-binding), corresponding to substitution for L_0, and a single-variable context, *i.e.*, only one variable can be in scope at a time. L_0 has terms $e ::= \mathbf{let}\ e\ \mathbf{in}\ e \mid \mathbf{var}$, where $\mathbf{var}$ denotes the single variable. Typing judgments have the form $\sigma \vdash$

$e : \tau$, where σ is the type of the single-variable context. The categorical semantics is defined inductively on the structure of type judgments as follows (which also shows the typing rules):

$$[\![\tau \vdash \mathbf{var} : \tau]\!] = id_\tau : [\![\tau]\!] \to [\![\tau]\!] \qquad \frac{[\![\tau_0 \vdash e_1 : \sigma]\!] = f : [\![\tau_0]\!] \to [\![\sigma]\!] \quad [\![\sigma \vdash e_2 : \tau]\!] = g : [\![\sigma]\!] \to [\![\tau]\!]}{[\![\tau_0 \vdash \mathbf{let}\ e_1\ \mathbf{in}\ e_2 : \tau]\!] = g \circ f : [\![\tau_0]\!] \to [\![\tau]\!]}$$

Thus L_0 is essentially the "pointed" version of point-free morphism-composition terms.

Properties. Since *let*-binding is modelled by composition, and variable access by identity, the axioms of a category imply the following equational theory for L_0:

$$\begin{array}{lrcl} (\textit{associativity}) & \mathbf{let}\ e_1\ \mathbf{in}\ (\mathbf{let}\ e_2\ \mathbf{in}\ e_3) & \equiv & \mathbf{let}\ (\mathbf{let}\ e_1\ \mathbf{in}\ e_2)\ \mathbf{in}\ e_3 \\ (\textit{right unit}) & \mathbf{let}\ \mathbf{var}\ \mathbf{in}\ e & \equiv & e \\ (\textit{left unit}) & \mathbf{let}\ e\ \mathbf{in}\ \mathbf{var} & \equiv & e \end{array}$$

Logic. A category with no additional structure also provides a model for a simple logical calculus $\mathcal{L}_0$ of hypothetical judgment, with inference rules:

$$\frac{A \vdash B \quad B \vdash C}{A \vdash C} \qquad \frac{}{A \vdash A} \tag{11}$$

Since $\mathcal{L}_0$ lacks implication there is no *deduction theorem* mapping a formulae under a hypothesis to an implication with no hypotheses. Instead (11) provides a kind of transitivity/cut-elimination rule, subsuming a deduction theorem (implication introduction, in natural deduction form) followed by modus ponens (implication elimination).

Logics corresponding to type theories or categories are not discussed further in this chapter.

2.2.2. Multi-variable contexts: products

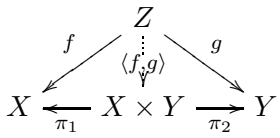
With only a single free variable in scope at any one time, L_0 is not particularly useful or practical for programming. L_1 extends L_0 with multi-variable environment, with syntax $e ::= \mathbf{let}\ v = e\ \mathbf{in}\ e \mid v$ where v ranges over syntactic variables. Similarly to λ^τ, typing judgments for L_1 have a context of free variables associating types to variables, with rules:

$$[\text{let}_1]\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, v : \sigma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \tau} \qquad [\text{var}_1]\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau} \tag{12}$$

A categorical semantics for L_1 requires a categorical model for the concatenation of free-variable typing assumptions and for empty contexts. Multiple assumptions are modelled by categories with *product* objects and empty contexts by *terminal* objects.

Definition 2.2.3. A category $\mathbb{C}$ has *products* (also called *categorical* or *Cartesian products*) if for all $X, Y \in \mathbb{C}_0$ there is an object $X \times Y \in \mathbb{C}_0$ and morphisms $\pi_1 : X \times Y \to X, \pi_2 : X \times Y \to Y \in \mathbb{C}_1$ (called *projections*), such that for any two morphisms $f : Z \to X$, $g : Z \to Y \in \mathbb{C}_1$ there is a unique morphism called the *pairing* of f and g, denoted $\langle f, g\rangle : Z \to X \times Y \in \mathbb{C}_1$, making

the following diagram commute (the *universal property* of products) [ML98]:



(13)

Products can be used for the semantics of multi-variable contexts, where the categorical semantics of L_1 maps terms to morphisms whose source is a product of free-variable assumptions:

$$[\![\Gamma \vdash e : \tau]\!] : ([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!] \in \mathbb{C}_1 \quad \textit{where } \Gamma = v_1 : \tau_1, \ldots, v_n : \tau_n$$

For brevity, interpretations of a context Γ are sometimes written as just $[\![\Gamma]\!]$ and interpretation brackets $[\![-]\!]$ are sometimes elided from the objects of a morphism to reduce clutter in the rules.

The *empty context* can be modelled as a terminal object $1 \in \mathbb{C}_0$ in the category.

Definition 2.2.4. A category $\mathbb{C}$ has a terminal object $1 \in \mathbb{C}_0$ if for every object $A \in \mathbb{C}_0$ there is a unique morphism $!_A : A \to 1 \in \mathbb{C}_1$. Therefore $!_B \circ f = !_X \circ g$ for $f : A \to B, g : A \to X \in \mathbb{C}_1$.

The semantics of L_1 interprets *let*-binding as composition and *pairing*, passing the incoming context Γ to both expressions, and variable access as projection:

$$\frac{v_i : \tau_i \in \Gamma \quad |\Gamma| = n \quad 1 \leqslant i \leqslant n}{[\![\Gamma \vdash v_i : \tau_i]\!] = \pi_i^n : \Gamma \to \tau_i} \qquad \frac{[\![\Gamma \vdash e_1 : \sigma]\!] = k_1 : \Gamma \to \sigma \quad [\![\Gamma, v : \sigma \vdash e_2 : \tau]\!] = k_2 : \Gamma \times \sigma \to \tau}{[\![\Gamma \vdash \textbf{let } v = e_1 \textbf{ in } e_2]\!] = k_2 \circ \langle id, k_1 \rangle : \Gamma \to \tau}$$

where products here are assumed to be left-associated, *e.g.*, $(\ldots \times \tau_{n-1}) \times \tau_n$, and π_i^n is defined:

$$\pi_i^n = \pi_2 \circ {\pi_1}^{(n-i)} : (((1 \times X_1) \ldots \times X_i) \times \ldots X_n) \to X_i \tag{14}$$

i.e., the i^{th} projection from an n-product, where ${\pi_1}^k$ is the k-times repeated composition of π_1.

Properties. From the laws of a category with products, L_1 has the following equational theory:

(*associativity*)	$\textbf{let } x{=}e_1 \textbf{ in } (\textbf{let } y{=}e_2 \textbf{ in } e_3)$	$\equiv$	$\textbf{let } y{=}(\textbf{let } x{=}e_1 \textbf{ in } e_2) \textbf{ in } e_3$	*iff* $x \notin FV(e_3)$
(*left unit*)	$\textbf{let } x = e \textbf{ in } x$	$\equiv$	e	
(*right unit*)	$\textbf{let } x = y \textbf{ in } e$	$\equiv$	$e[x := y]$	
(*weaken*)	$\textbf{let } x = e_1 \textbf{ in } e_2$	$\equiv$	e_2	*iff* $x \notin FV(e_2)$
(*contraction*)	$\textbf{let } x = e \textbf{ in } (\textbf{let } y = e \textbf{ in } e')$	$\equiv$	$\textbf{let } x = e \textbf{ in } e'[y := x]$	
(*exchange*)	$\textbf{let } x{=}e_1 \textbf{ in } (\textbf{let } y{=}e_2 \textbf{ in } e_3)$	$\equiv$	$\textbf{let } y{=}e_2 \textbf{ in } (\textbf{let } x{=}e_1 \textbf{ in } e_3)$	*iff* $x \notin FV(e_2)$, $y \notin FV(e_1)$

The following lemma provides the (*let-*β) property, which can be proved by induction over the structure of type derivations (the proof is omitted here for brevity):

Lemma 2.2.5. *(let-*β*) (***let** *equivalent to substitution)*

$$\forall e, e'.\ (\Gamma, x : \tau' \vdash e : \tau) \wedge (\Gamma \vdash e' : \tau') \Rightarrow [\![\Gamma \vdash \textbf{let } x = e' \textbf{ in } e : \tau]\!] = [\![\Gamma \vdash e[x := e'] : \tau]\!]$$

2.2.3. Structural rules: monoidal categories

Structural typing rules of the simply-typed λ-calculus are admissible (*i.e.*, can be elided) for typing, however proving the above equations for **let** requires structural rules to be included in

the semantics. For example, weakening has semantics:

$$[\text{WEAKEN}]\frac{[\![\Gamma \vdash e : \tau]\!] = k : [\![\Gamma]\!] \to [\![\tau]\!]}{[\![\Gamma, x : \tau' \vdash e : \tau]\!] = k \circ \pi_1 : [\![\Gamma]\!] \times [\![\tau']\!] \to [\![\tau]\!]}$$

Proof of the above (*weaken*) rule is then as follows (where $x \notin FV(e_2)$):

$$\begin{aligned}
[\![\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau_2]\!] &= [\![\Gamma, x : \tau_1 \vdash e_2 : \tau_2]\!] \circ \langle id, [\![\Gamma \vdash e_1 : \tau_1]\!] \rangle && [\![\textbf{let}]\!] \\
&= [\![\Gamma \vdash e_2 : \tau_2]\!] \circ \pi_1 \circ \langle id, [\![\Gamma \vdash e_1 : \tau_1]\!] \rangle && [\![\text{WEAKEN}]\!] \\
&= [\![\Gamma \vdash e_2 : \tau_2]\!] && \{\times \textit{ universal property}\} \quad \square
\end{aligned}$$

The (*exchange*) rule requires an operation for reassociation and permutation which can be provided by a natural transformation $\chi_{A,B,C} : (A \times B) \times C \to (A \times C) \times B$:

$$\chi_{A,B,C} = \langle \pi_{1_{A,B}} \times id_C, \pi_{2_{A,B}} \circ \pi_{1_{(A \times B),C}} \rangle \tag{15}$$

The semantics of exchange can be defined by lifting χ using the $(- \times -)$ bifunctor:

$$[\text{EXCHANGE}]\frac{[\![\Gamma, x : \sigma, y : \tau, \Gamma' \vdash e : \tau]\!] = f : ((([\![\Gamma]\!] \times [\![\sigma]\!]) \times [\![\tau]\!]) \times [\![\Gamma']\!]) \to [\![\tau]\!]}{[\![\Gamma, y : \tau, x : \sigma, \Gamma' \vdash e : \tau]\!] = f \circ (\chi_{\Gamma,\sigma,\tau} \times id) : ((([\![\Gamma]\!] \times [\![\tau]\!]) \times [\![\sigma]\!]) \times [\![\Gamma']\!]) \to [\![\tau]\!]}$$

Similarly, the semantics of the remaining structural rules is provided by products, pairing, and projections. In general, these notions are captured by a *symmetric Cartesian monoidal category*:

Definition 2.2.6. [ML98] A *monoidal category* $(\mathbb{C}, \otimes, I)$ comprises a category $\mathbb{C}$ along with a *bifunctor* $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$, a unit object $I \in \mathbb{C}_0$, and three natural isomorphisms:

$$\begin{array}{ll}
(\textit{associativity}) & \alpha_{A,B,C} : (A \otimes B) \otimes C \cong A \otimes (B \otimes C) \\
(\textit{left unit}) & \lambda_A : (I \otimes A) \cong A \\
(\textit{right unit}) & \rho_A : (A \otimes I) \cong A
\end{array}$$

such that associativity and units commute with each other, *e.g.*, $(id_X \otimes \lambda_Y) \circ \alpha_{X,I,Y} = \rho_X \otimes id_Y$. Diagrams of the coherence conditions can be found in Appendix B.1.2 (p. 185).

Definition 2.2.7. A monoidal category $(\mathbb{C}, \otimes, I)$ is *symmetric* if it has the natural isomorphism:

$$(\textit{symmetry}) \quad \gamma_{A,B} : (A \otimes B) \cong (B \otimes A)$$

such that γ commutes with α, λ and ρ (coherence conditions in Appendix B.1.2).

Example 2.2.8. A category $\mathbb{C}$ with products $\times$ and terminal object 1 is symmetric monoidal, $(\mathbb{C}, \times, 1)$, with the associativity, unit, and symmetry isomorphisms defined in terms of pairing, projections, and terminal morphisms (definitions given in Appendix B.1.3, (p. 186)). The coherence conditions for the monoidal category and the isomorphism properties follow straightforwardly from the universal properties of products and terminal morphisms.

Definition 2.2.9. A *Cartesian monoidal category* (alternatively, *Cartesian category*) is a monoidal category where every object $X \in \mathbb{C}_0$ has the trivial duplication *comonoid*, with $\Delta_X : X \to X \otimes X$ and $e_X : X \to I$ such that $\lambda_X \circ (e_X \otimes id) \circ \Delta_X = \rho_X \circ (id \otimes e_X) \circ \Delta_x$.

Alternatively, a Cartesian category may be defined as a monoidal category whose bifunctor $(- \otimes -)$ satisfies the properties of a Cartesian product, with pairing $\langle f, g\rangle = (f \otimes g) \circ \Delta$ and projections $\pi_1 = \rho_X \circ (id \otimes e_X)$, $\pi_2 = \lambda_X \circ (e_X \otimes id)$.

The operations of a symmetric Cartesian monoidal category thus provide the semantics of weakening (via projections), exchange (via a combination of associativity and symmetry), and contraction (via Δ). Section 2.2.7 summarises the semantics of the structural rules.

2.2.4. Abstraction: exponents and adjunctions

The L_1 language resembles that of the simply-typed λ-calculus but lacks the crucial feature of *abstraction* and function types. Function types are modelled by *exponent objects*.

Definition 2.2.10. A category $\mathbb{C}$ with products has *exponents* where for all $X, Y \in \mathbb{C}_0$ there is an object $X \Rightarrow Y \in \mathbb{C}_0$ and a morphism $\mathsf{ev} : (X \Rightarrow Y) \times X \to Y \in \mathbb{C}_1$ such that for all $g : X \times Y \to Z \in \mathbb{C}_1$ there is a unique morphism denoted $\lambda g : X \to (Y \Rightarrow Z) \in \mathbb{C}_1$ making the following diagram commute (the *universal property* of exponents) [**ML98**]:

$$\begin{array}{ccc} X \times Y & & \\ {\scriptstyle \lambda g \times id_Y}\,\vdots & \searrow^{g} & \\ (Y \Rightarrow Z) \times Y & \xrightarrow[\mathsf{ev}]{} & Z \end{array} \tag{16}$$

Definition 2.2.11. A *Cartesian closed category* (CCC) is a category which has products and exponents (satisfying the universal properties) for all possible pairs of objects.

The semantics of application and abstraction can then be defined in terms of λ and ev as:

$$[\text{ABS}]\frac{[\![\Gamma, x : \sigma \vdash e : \tau]\!] = k : \Gamma \times \sigma \to \tau}{[\![\Gamma \vdash \lambda x.e : \sigma \Rightarrow \tau]\!] = \lambda k : \Gamma \to (\sigma \Rightarrow \tau)}$$

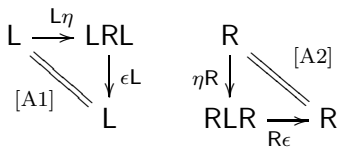
$$[\text{APP}]\frac{[\![\Gamma \vdash e_1 : \sigma \to \tau]\!] = k_1 : \Gamma \to (\sigma \Rightarrow \tau) \qquad [\![\Gamma \vdash e_2 : \sigma]\!] = k_2 : \Gamma \to \sigma}{[\![\Gamma \vdash e_1\, e_2 : \tau]\!] = \mathsf{ev} \circ \langle k_1, k_2\rangle : \Gamma \to \tau}$$

Properties. $\beta\eta$-equality can now be provided for the calculus (see Section 2.1.2). Appendix C.1.2 (p. 195) shows the proofs which follow from the universal properties of products and exponents. For $[\equiv\text{-}\beta]$, $[\equiv\text{-}let\text{-}\lambda]$ is proved first (*i.e.*, $(\lambda x.e_1)\, e_2 = \mathbf{let}\ x = e_2\ \mathbf{in}\ e_1$) then the proof relies on $[let\text{-}\beta]$ (Lemma 2.2.5, p. 28), showing **let** equivalent to syntactic substitution.

Adjunctions. Cartesian closed categories can be given an alternate, but equivalent, definition in terms of *adjunctions*. This presentation in terms of adjunctions aids later developments. An adjunction describes a relationship between two functors, with several equivalent definitions. Two definitions, called the *hom-set adjunction* and the *counit-unit adjunction* are used here.

Definition 2.2.12 (*Unit-counit adjunction*)**.** For categories $\mathbb{C}$ and $\mathbb{D}$, and a pair of functors $\mathsf{L} : \mathbb{D} \to \mathbb{C}$, $\mathsf{R} : \mathbb{C} \to \mathbb{D}$, L is *left adjoint* to R (conversely R is *right adjoint* to L), denoted $\mathsf{L} \dashv \mathsf{R}$, if there exists two natural transformations $\eta : 1_{\mathbb{D}} \to \mathsf{RL}$ and $\epsilon : \mathsf{LR} \to 1_{\mathbb{C}}$ called the *unit* and

counit of the adjunction respectively, satisfying [A1] and [A2]:



Example 2.2.13. Products are left-adjoint to exponents $(- \times X) \dashv (X \Rightarrow -)$, where:

$$\eta : 1 \to (X \Rightarrow -)(- \times X) \qquad \epsilon : (X \times -)(X \Rightarrow -) \to 1$$
$$\eta\ a = \lambda x.(a, x) \qquad \epsilon\ (f, n) = fn$$

Definition 2.2.14 (*Hom-set adjunction*)**.** For categories $\mathbb{C}$ and $\mathbb{D}$, and functors $\mathsf{L} : \mathbb{D} \to \mathbb{C}$, $\mathsf{R} : \mathbb{C} \to \mathbb{D}$ then $\mathsf{L} \dashv \mathsf{R}$ if there exists a family of bijections (for all objects $X \in \mathbb{C}_0$ and $Y \in \mathbb{D}_0$):

$$\phi_{Y,X} : \hom_{\mathbb{C}}(\mathsf{L}Y, X) \cong \hom_{\mathbb{D}}(Y, \mathsf{R}X)$$

natural in Y and X (ϕ is a *natural isomorphism*); ϕ is the *left adjunct* and ϕ^{-1} the *right adjunct*.

Example 2.2.15. For $(- \times X) \dashv (X \Rightarrow -)$, the hom-set adjunction is provided by *currying* and *uncurrying*: $\mathsf{curry}/\mathsf{uncurry} : (A \times X \to B) \cong (A \to (X \Rightarrow B))$.

Lemma 2.2.16. *For functors* $\mathsf{L} : \mathbb{D} \to \mathbb{C}$, $\mathsf{R} : \mathbb{C} \to \mathbb{D}$ *where* $\mathsf{L} \dashv \mathsf{R}$ *with* $\epsilon : \mathsf{LR} \to 1_{\mathbb{C}}$ *and* $\eta : 1_{\mathbb{D}} \to \mathsf{RL}$ *there is an equivalent hom-set adjunction* $\phi : hom_{\mathbb{D}}(\mathsf{L}Y, X) \cong hom_{\mathbb{C}}(Y, \mathsf{R}X)$ *where:*

$$\begin{aligned} \phi\ f &= \mathsf{R}f \circ \eta \quad (\forall f : \mathsf{L}A \to B) \\ \phi^{-1}\ g &= \epsilon \circ \mathsf{L}g \quad (\forall g : A \to \mathsf{R}B) \end{aligned} \tag{17}$$

The proof of this construction is omitted for the sake of brevity.

Lemma 2.2.17. *If* $(- \times X) \dashv (X \Rightarrow -)$*, then all* $X \Rightarrow Y$ *are categorical exponents where* $\mathsf{ev} = \epsilon$*,* $\lambda f = \phi f$ *(proof in Appendix C.1.3, p.196).*

The semantics of abstraction and application for the simply-typed λ-calculus can therefore be defined in terms of the adjunction for products and exponents:

$$[\text{ABS}]\frac{[\![\Gamma, x : \sigma \vdash e : \tau]\!] = k : \Gamma \times \sigma \to \tau}{[\![\Gamma \vdash \lambda x.e : \sigma \Rightarrow \tau]\!] = \phi k : \Gamma \to (\sigma \Rightarrow \tau)}$$

$$[\text{APP}]\frac{[\![\Gamma \vdash e_1 : \sigma \to \tau]\!] = k_1 : \Gamma \to (\sigma \Rightarrow \tau) \qquad [\![\Gamma \vdash e_2 : \sigma]\!] = k_2 : \Gamma \to \sigma}{[\![\Gamma \vdash e_1\ e_2 : \tau]\!] = \phi^{-1} k_1 \circ \langle id, k_2 \rangle : \Gamma \to \tau}$$

Proofs of β- and η-equivalence therefore follow from the properties of adjunctions, where $[\equiv\text{-}\beta]$ is proved, as before, first by proving $[\equiv\text{-}let\text{-}\lambda]$ and then applying $[let\text{-}\beta]$, Lemma 2.2.5 (p.28):

$$\begin{aligned} & [\![\Gamma \vdash (\lambda x.\, e_1)\, e_2 : \tau]\!] \\ = \ & \phi^{-1}(\phi\, [\![\Gamma, x : \sigma \vdash e_1 : \tau]\!]) \circ \langle id, [\![\Gamma \vdash e_2 : \sigma]\!] \rangle && [\![\text{APP}]\!], [\![\text{ABS}]\!] \\ = \ & [\![\Gamma, x : \sigma \vdash e_1 : \tau]\!] \circ \langle id, [\![\Gamma \vdash e_2 : \sigma]\!] \rangle && \{\phi^{-1} \circ \phi = id\} \\ = \ & [\![\Gamma \vdash e_1[x := e_2] : \tau]\!] && [\![\mathbf{let}]\!], [let\text{-}\beta] \text{ - Lemma 2.2.5} \quad \square \end{aligned} \tag{18}$$

$$
\begin{array}{rll}
 & [\![\Gamma \vdash \lambda x.f\, x : \sigma \Rightarrow \tau]\!] & \\
 & = \phi\,(\phi^{-1}\,[\![\Gamma, x:\sigma \vdash f : \tau \to \tau]\!]) \circ \langle id, [\![\Gamma, x:\sigma \vdash x:\sigma]\!]\rangle) & [\![\textsc{abs}]\!], [\![\textsc{app}]\!] \\
 & = \phi\,(\phi^{-1}\,([\![\Gamma \vdash f : \sigma \Rightarrow \tau]\!] \circ \pi_1) \circ \langle id, \pi_2\rangle) & [\![\textsc{weaken}]\!], [\![\textsc{var}]\!] \\
(*) & = \phi\,(\phi^{-1}\,[\![\Gamma \vdash f : \sigma \Rightarrow \tau]\!]) \circ (\pi_1 \times id) \circ \langle id, \pi_2\rangle) & \{\phi^{-1}(g \circ f) = (\phi^{-1}g) \circ \mathsf{L}f\} \\
 & = \phi\,(\phi^{-1}\,[\![\Gamma \vdash f : \sigma \Rightarrow \tau]\!]) & \{\times \text{ universal property}\} \\
 & = [\![\Gamma \vdash f : \sigma \Rightarrow \tau]\!] & \{\phi \circ \phi^{-1} = id\} \quad \square
\end{array}
\tag{19}
$$

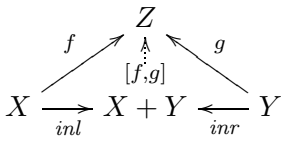
Note that the proof of $[\equiv\text{-}\beta]$ uses the property that $\phi^{-1} \circ \phi = id$ and $[\equiv\text{-}\eta]$ that $\phi \circ \phi^{-1} = id$. The proof of $[\equiv\text{-}\eta]$ at (*) uses a derived property of the adjunction following from the construction in Lemma 2.2.16. Alternatively, the proof could have used the dual derived property $\phi\,(g \circ f) = \mathsf{R}g \circ \phi f$ at (*), with an alternative but equivalent proof proceeding from (*). In a situation without full $\beta\eta$-equivalence, *i.e.*, without an adjunction, some of these axioms used here may still hold, providing some other weaker equational theory. Section 2.2.7 summarises the categorical semantics here, turning these axioms into requirements for different equational theories. Throughout, ψ is used to denote a natural transformation of the same type as ϕ^{-1}, but which is not necessarily the inverse of ϕ.

An adjunction induces a monad and comonad. Conversely any comonad or monad induces an adjunction [**EM65**]. This aspect of comonads will not feature strongly in this dissertation.

2.2.5. Sum types: coproducts

The extension of the simply-typed λ-calculus with values of product and sum types is straightforward, reusing the product structure used for contexts, and adding finite coproducts.

Definition 2.2.18. A category $\mathbb{C}$ has *coproducts* if for all $X, Y \in \mathbb{C}_0$ there is an object $X+Y \in \mathbb{C}_0$ and *injection* morphisms $inl : X \to X+Y, inr : Y \to X+Y \in \mathbb{C}_1$, such that for all $f : X \to Z$, $g : Y \to Z \in \mathbb{C}_1$ there is a unique *copairing* of f and g, denoted $[f,g] : X+Y \to Z \in \mathbb{C}_1$ (analogous to a *case* expression), making the following diagram commute [**ML98**]:



(20)

2.2.6. Completeness

The preceding sections have focussed on *soundness* of the CCC model for the simply-typed λ-calculus: if two terms are equal with respect to $\beta\eta$-equality, then their denotations in a CCC are equal, *i.e.*, $\Gamma \vdash M \equiv_{\beta\eta} N : \tau$ implies $[\![\Gamma \vdash M : \tau]\!] = [\![\Gamma \vdash N : \tau]\!]$. The converse property is that of *completeness*, *i.e.*, where $[\![\Gamma \vdash M : \tau]\!] = [\![\Gamma \vdash N : \tau]\!]$ then $\Gamma \vdash M \equiv_{\beta\eta} N : \tau$. However, not all CCCs provide a complete model of simply-typed λ-calculus [**Sim95**].

Completeness of models is not addressed further in this dissertation. Instead, the focus is more often on soundness, where the equational theory of a calculus or language is supported by the categorical model such that it is sound.

$$[\![\text{ABS}]\!]\frac{[\![\Gamma, x:\sigma \vdash e:\tau]\!] = k : \Gamma \times \sigma \to \tau}{\begin{aligned}[\![\Gamma \vdash \lambda x.e : \sigma \Rightarrow \tau]\!] &= \lambda\, k : \Gamma \to (\sigma \Rightarrow \tau) \\ &= \phi\, k : \Gamma \to (\sigma \Rightarrow \tau)\end{aligned}}$$

$$[\![\text{APP}]\!]\frac{[\![\Gamma \vdash e_1 : \sigma \Rightarrow \tau]\!] = k_1 : \Gamma \to (\sigma \Rightarrow \tau) \qquad [\![\Gamma \vdash e_2 : \sigma]\!] = k_2 : \Gamma \to \sigma}{\begin{aligned}[\![\Gamma \vdash e_1\, e_2 : \tau]\!] &= \mathsf{ev} \circ \langle k_1, k_2\rangle : \Gamma \to \tau \\ &= \phi^{-1} k_1 \circ \langle id, k_2\rangle : \Gamma \to \tau\end{aligned}}$$

$$[\![\text{VAR}]\!]\frac{x_i : \tau_i \in \Gamma}{[\![\Gamma \vdash x_i : \tau]\!] = \pi_2 \circ \pi_1^{(n-i)} : \Gamma \to \tau_i} \quad \text{where } (|\Gamma| = n) \wedge (1 \leqslant i \leqslant n)$$

$$[\![\text{LET}]\!]\frac{[\![\Gamma \vdash e_1 : \sigma]\!] = k_1 : \Gamma \to \sigma \qquad [\![\Gamma, v:\sigma \vdash e_2 : \tau]\!] = k_2 : \Gamma \times \sigma \to \tau}{[\![\Gamma \vdash \textbf{let } v = e_1 \textbf{ in } e_2]\!] = k_2 \circ \langle id, k_1\rangle : \Gamma \to \tau}$$

(a) Core rules

$$[\![\text{CONTRACT}]\!]\frac{[\![\Gamma, y:\tau', z:\tau' \vdash e:\tau]\!] = k : (\Gamma \times \tau') \times \tau' \to \tau}{[\![\Gamma, x:\tau' \vdash e[z := x][y := x]]\!] = k \circ \alpha^{-1} \circ (id \times \Delta) : \Gamma \times \tau' \to \tau}$$

$$[\![\text{WEAKEN}]\!]\frac{[\![\Gamma \vdash e:\tau]\!] = k : \Gamma \to \tau}{[\![\Gamma, x:\tau' \vdash e:\tau]\!] = k \circ \pi_1 : \Gamma \times \tau' \to \tau}$$

$$[\![\text{EXCHANGE}]\!]\frac{[\![\Gamma, y:\tau, x:\sigma \vdash e:\tau]\!] = k : (\Gamma \times \tau) \times \sigma \to \tau}{[\![\Gamma, x:\sigma, y:\tau \vdash e]\!] = k \circ \alpha^{-1} \circ (id \times \gamma) \circ \alpha : (\Gamma \times \sigma) \times \tau \to \tau}$$

(b) Structural rules

Figure 2.3. Categorical semantics for the simply-typed λ-calculus in a CCC.

2.2.7. Summary

Cartesian-closed categories provide a $\beta\eta$-equivalence model of the simply-typed λ-calculus [**LS88**, **Lam80**]. **Figure** 2.3 collects the definitions given in the preceding sections for the categorical semantics of λ^τ, including the interpretation for application and abstraction in terms of both adjunctions, and the λ and ev constructions of exponents.

The following summarises the additional structure needed on a basic category $\mathbb{C}$ such that it provides a $\beta\eta$-equivalence model of the simply-typed λ-calculus. The presentation here separates these into *structural requirements (A.i)-(A.v)*, that provide the denotations of terms, and *coherence condition requirements* that lead to $\beta\eta$-equality for the calculus *(B.i)-(B.iv)*. This separation is useful for the categorical semantics of λ-calculus-like systems later in the dissertation which have the same semantic definitions as in **Figure** 2.3, but defined over categories where additional structure may be required for β- or η-equivalence.

Specifically a category $\mathbb{C}$ must be equipped with the following to be a ($\beta\eta$-equivalence) model of the simply-typed λ-calculus:

(A.i) Substitution – Associative and unital composition, automatic since $\mathbb{C}$ is a category;

(A.ii) *Free-variable contexts* – For all pairs of objects $A, B \in \mathbb{C}_0$ an object $A \times B \in \mathbb{C}_0$ with a notion of *pairing* $\langle f, g\rangle : A \to (X \times Y) \in \mathbb{C}_1$ for all $f : A \to X, g : A \to Y \in \mathbb{C}_1$, modelling the sharing of a context between two sub-terms, and *projections* $\pi_1 : A \times B \to A, \pi_2 : A \times B \to B \in \mathbb{C}_1$;

(A.iii) *Functions* – For all pairs of objects $A, B \in \mathbb{C}_0$ an object $A \Rightarrow B \in \mathbb{C}_0$;

(A.iv) *Abstraction* – Currying natural transformation $\phi_{A,B} : \mathbb{C}(A \times X, B) \to \mathbb{C}(A, X \Rightarrow B)$;

(A.v) *Application* – Uncurrying natural transformation $\psi_{A,B} : \mathbb{C}(A, X \Rightarrow B) \to \mathbb{C}(A \times X, B)$ (unabstraction);

(B.i) [$\equiv$-*let*-λ] – $\psi \circ \phi = id$ (*i.e.*, ψ is the left-inverse of ϕ);

(B.ii) [$\equiv$-β] – via inductive proof or [$\equiv$-*let*-λ] with an inductive proof of [*let*-β];

(B.iii) [$\equiv$-η] – $\phi \circ \psi = id$ (*i.e.*, ψ is the right-inverse of ϕ) and either:

$$\phi\,(g \circ f) = (id \Rightarrow g) \circ \phi f \tag{21}$$
$$\psi(g \circ f) = (\psi g) \circ (f \times id) \tag{22}$$

where $(- \times -)$ and $(- \Rightarrow -)$ are bifunctors;

(B.iv) *Structural rules* – $(\mathbb{C}, \times, 1)$ is a symmetric monoidal category;

Conditions *(B.i)*-*(B.iv)* are equivalent to requiring that $\times$ is a categorical product and $\Rightarrow$ is a categorical exponent (*i.e.*, that $\mathbb{C}$ is Cartesian closed). If the product and exponent structures of $\mathbb{C}$ are adjoint (*i.e.*, $(- \times A) \dashv (A \Rightarrow -)$) then *(B.i)*-*(B.iii)* are automatic [**LS88**].

2.3. Strong monads and the effectful λ-calculus

For the pure simply-typed λ-calculus, equality implies reduction, *i.e.*, if two terms are equal in the equational theory then one can be reduced into the other, and conversely, if a term reduces into another then the two terms are equal in the equational theory. However, in the presence of *side effects*, such as state and non-termination, $\beta\eta$-reduction does not always preserve equality [**Mog89**]. For example, consider the non-terminating computation $\bot$ (*i.e.*, $\bot$ has an infinite reduction sequence $\bot \rightsquigarrow \bot \rightsquigarrow \ldots$) which can be given any type, *i.e.*, $\Gamma \vdash \bot : \tau$. By η-reduction $\lambda x.\bot\, x \rightsquigarrow_\eta \bot$, however $(\lambda x.\bot x) \not\equiv \bot$ (assuming call-by-value semantics) since the LHS is a terminating function value and the RHS is non-terminating. Thus, for effectful computations a different (restricted) equational theory holds than that of the simply-typed λ-calculus.

Rather than defining the semantics of an effectful language for each possible notion of effect *ad hoc*, Moggi showed that various notions of impure computation, such as state, non-determinism, continuations, and partiality (exceptions/non-termination), are captured by the structure of a *(strong) monad* [**Mog89**, **Mog91**]. Moggi defines a general calculus of computations λ_c requiring only the definition of a particular strong monad for the required notion of effect. Denotations of the λ_c are morphisms $\sigma \to \mathsf{M}\,\tau$ where M is an endofunctor encoding impure computations of pure values of type τ. For example, $\mathsf{M}A = A + \{\bot\}$ models partial computations. A compositional semantics is provided if M has the additional structure of a (strong) monad.

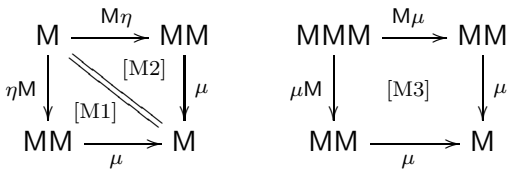
2.3.1. Monads

Definition 2.3.1. A *monad* is a triple (M, η, μ) of:

- an endofunctor $\mathsf{M} : \mathbb{C} \to \mathbb{C}$;
- a natural transformation $\eta : 1_{\mathbb{C}} \dot{\to} \mathsf{M}$ called the *unit*;
- a natural transformation $\mu : \mathsf{MM} \dot{\to} \mathsf{M}$ called the *multiplication*.

with the following axioms [M1-3]:

$$\begin{array}{ll} [\text{M1}] & \mu \circ \eta\mathsf{M} = 1_{\mathsf{M}} \\ [\text{M2}] & \mu \circ \mathsf{M}\eta = 1_{\mathsf{M}} \\ [\text{M3}] & \mu \circ \mu\mathsf{M} = \mu \circ \mu\mathsf{M} \end{array}$$



From the perspective of effectful computations, the unit operation constructs a trivially effectful computation for a value and multiplication reduces a 1-nested computation into a single computation, combining, or composing, the effects of the inner and outer computations. The axioms of a monad imply that the encoded effect information is composed *monoidally*. Indeed, monads are themselves monoids defined on the category of endofunctors $[\mathbb{C}, \mathbb{C}]$, which has endofunctors as objects, natural transformations as morphisms, and where $[\mathbb{C}, \mathbb{C}]$ is a monoidal category with the monoidal structure provided by composition of functors with $([\mathbb{C}, \mathbb{C}], \circ, 1_{\mathbb{C}})$.

The operations of a monad provide (associative and unital) composition for morphisms $\sigma \to \mathsf{M}\tau$. Thus a monad induces a category, called the *Kleisli category* [**ML98**].

Definition 2.3.2. [**ML98**] Given a monad M on a category $\mathbb{C}$, the *Kleisli category* $\mathbb{C}_{\mathsf{M}}$, has objects $\mathbb{C}_{\mathsf{M}0} = \mathbb{C}_0$ and, for all objects $A, B \in \mathbb{C}_0$, morphisms $\mathbb{C}_{\mathsf{M}}(A, B) = \mathbb{C}(A, \mathsf{M}B)$, with:

- Composition: for all $g \in \mathbb{C}_{\mathsf{M}}(B, C)$, $f \in \mathbb{C}_{\mathsf{M}}(A, B)$, then $g \mathbin{\hat{\circ}} f = \mu \circ \mathsf{M}g \circ f \in \mathbb{C}_{\mathsf{M}}(A, C)$
- Identities: $\hat{id}_A = \eta_A \in \mathbb{C}_{\mathsf{M}}(A, A)$

A Kleisli category therefore captures computations of an effectful language where composition composes the resulting effects from two computations. Functions/morphisms of type $\tau \to \mathsf{M}\tau'$ are referred to as *Kleisli morphisms* of the monad M.

Monads have an equivalent presentation known as the *Kleisli triple form*.

Definition 2.3.3. The *Kleisli triple* of a monad comprises an object mapping $\mathsf{M} : \mathbb{C}_0 \to \mathbb{C}_0$, an *extension* operation $(-)^\dagger$ such that for all $f : A \to \mathsf{M}B$ then $f^\dagger : \mathsf{M}A \to \mathsf{M}B$ (natural in A and B), and unit operation $\eta_A : A \to \mathsf{M}A$, satisfying [K1-3] for all $f : X \to \mathsf{M}Y, g : Y \to \mathsf{M}Z \in \mathbb{C}_1$:

$$[\text{K1}] \quad \eta_X^* = id_{\mathsf{M}X} \qquad [\text{K2}] \quad f^* \circ \eta_Y = f \qquad [\text{K3}] \quad (g^* \circ f)^* = g^* \circ f^*$$

The Kleisli triple and monoidal form of a monad (in terms of μ) are equivalent by the equalities:

$$f^* = \mu \circ \mathsf{M}f \qquad \mu = id_{\mathsf{M}}{}^* \qquad \mathsf{M}f = (\eta \circ f)^* \tag{23}$$

Thus, composition in a Kleisli category can be alternatively defined: $g \mathbin{\hat{\circ}} f = g^* \circ f$.

Examples 2.3.4 The following examples shows common monads in Kleisli triple form, where the operations are defined using the syntax of the λ-calculus with products and coproducts:

- *Non-termination, partiality, or exception, monad*, $\mathsf{M}A = A+1$ (written $\mathsf{M}A = A+\{\bot\}$ before)
  - $\mathsf{M}(f : A \to B) = [f, id] \quad : A + 1 \to B + 1$
  - $\eta = inl$ (pure computations of a value are total)
  - $k^* = [[k \circ inl, inr], inr]$ (if either the incoming or returned computation is undefined then the result is undefined).
- *Exponent monad*, $\mathsf{M}A = X \Rightarrow A$, captures computations with an implicit parameter, defined:
  - $\mathsf{M}\,(f : A \to B)x \;= f \circ x \quad : (X \Rightarrow A) \to (X \Rightarrow B)$
  - $\eta\; a = \lambda x.a$ (pure computations of a value ignore implicit parameters)
  - $k^*\, e = \lambda x.((k\, e)\; x)\; x$ (composed computations are each passed the same parameter)
- *State monad*, $\mathsf{M}A = S \Rightarrow (A \times S)$ for some type of states S, captures computations which take a state and return a result and a new state (*cf.* Kleisli morphisms of the state monad $A \to (S \Rightarrow (B \times S))$ with transitions of an operational semantics $\langle e, s\rangle \to \langle e', s'\rangle$):
  - $\mathsf{M}\,(f : A \to B)\; x = (f \times id)\, x \quad : (S \Rightarrow (A \times S)) \to (S \Rightarrow (B \times S))$
  - $\eta\; a = \lambda s.(a, s)$ (pure computations return incoming state unchanged)
  - $k^*\; e = \lambda s.\mathbf{let}\; (a, s') = e\; s\; \mathbf{in}\; (k\, a)s'$ (composed computations have the state threaded through the incoming and result computations)

Strong monads. Strong monads provide additional structure required for handling contexts in the simply-typed λ-calculus, shown in Section 2.3.2.

Definition 2.3.5. A *strong monad* comprises a monad (M, μ, η) and a *tensorial strength* $\mathsf{st}_{A,B} : (A \times \mathsf{M}B) \to \mathsf{M}(A \times B)$, satisfying a number of coherence conditions between the operations of the monad and st (elided for brevity here, but included in Appendix B.1.4, p. 188).

For a functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ with tensorial strength, if $\mathbb{C}$ is a *symmetric* monoidal category, with $\gamma : A \otimes B \to B \otimes A$, then there is a symmetric notion:

$$\mathsf{st}'_{A,B} = \; \mathsf{F}A \otimes B \xrightarrow{\gamma} B \otimes \mathsf{F}A \xrightarrow{\mathsf{st}_{A,B}} \mathsf{F}(B \otimes A) \xrightarrow{\mathsf{F}\gamma} \mathsf{F}(A \otimes B)$$

2.3.2. Categorical semantics of λ_c

The semantics of λ_c, for a particular notion of effect M has the Kleisli category $\mathbb{C}_{\mathsf{M}}$ as its domain with interpretation of types $[\![\tau]\!] \in (\mathbb{C}_{\mathsf{M}})_0$ and well-typed terms:

$$[\![\overline{v : \tau} \vdash e : \tau]\!] : [\![\overline{\tau}]\!] \to [\![\tau]\!] \in (\mathbb{C}_{\mathsf{M}})_1$$
$$: [\![\overline{\tau}]\!] \to \mathsf{M}[\![\tau]\!] \in \mathbb{C}_1$$

Moggi's λ_c is essentially a call-by-value simply-typed λ-calculus, although with some changes to the equational theory, in particular η-equivalence does not hold, as discussed above. The semantics here proceeds from the semantics of the simply-typed λ-calculus presented in Section 2.2.7, with structures that give the denotations of terms separated from axioms on these structures which give the equational laws of the λ-calculus.

For $\mathbb{C}_{\mathsf{M}}$ to provide a suitable model of λ_c some additional structure is required. Firstly, assume that the base category $\mathbb{C}$ is a CCC with products $\times$ and exponents $\Rightarrow$ and hom-set adjunction ϕ. The requirements on $\mathbb{C}_{\mathsf{M}}$ are then:

(A.i) Substitution – Associative and unital composition:
- Well-behaved composition is automatic since $\mathbb{C}_{\mathsf{M}}$ is a category.

(A.ii) Free-variable contexts – For all pairs of objects $A, B \in \mathbb{C}_{\mathsf{M}0}$ an object $A \times^{\mathsf{M}} B \in \mathbb{C}_{\mathsf{M}0}$ with a *pairing* $\langle f, g\rangle^{\mathsf{M}} : A \to (X \times^{\mathsf{M}} Y) \in \mathbb{C}_{\mathsf{M}1}$ and projections $\pi_1^{\mathsf{M}} : A \times^{\mathsf{M}} B \to A, \pi_2^{\mathsf{M}} : A \times^{\mathsf{M}} B \to B \in \mathbb{C}_{\mathsf{M}1}$:
- The tensor $\times^M$ in $\mathbb{C}_{\mathsf{M}}$ can be defined by products in $\mathbb{C}$, *i.e.*, $(A \times^M B) \in \mathbb{C}_{\mathsf{M}0} = (A \times B) \in \mathbb{C}_0$. Pairing two morphisms $f : A \to \mathsf{M}X, g : A \to \mathsf{M}Y$ is thus the morphism $\langle f, g\rangle^{\mathsf{M}} : A \to \mathsf{M}(X \times Y)$ defined for strong monads as follows:

$$\langle f, g\rangle^{\mathsf{M}} := A \xrightarrow{\langle f,g\rangle} \mathsf{M}X \times \mathsf{M}Y \xrightarrow{\mathsf{st}} \mathsf{M}(\mathsf{M}X \times Y) \xrightarrow{\mathsf{Mst}'} \mathsf{MM}(X \times Y) \xrightarrow{\mu} \mathsf{M}(X \times Y) \tag{24}$$

  with projections $\pi_1^{\mathsf{M}} = \eta \circ \pi_1 : A \times B \to \mathsf{M}A$ and $\pi_2^{\mathsf{M}} = \eta \circ \pi_2 : A \times B \to \mathsf{M}B$. Note, that $\times^M$ is not a categorical product.

(A.iii) Functions – For all objects $A, B \in \mathbb{C}_{\mathsf{M}0}$ an object $(A \Rightarrow^{\mathsf{M}} B) \in \mathbb{C}_{\mathsf{M}0}$;
- where $(\Rightarrow^{\mathsf{M}} B) \in \mathbb{C}_{\mathsf{M}0} = (A \Rightarrow \mathsf{M}B) \in \mathbb{C}_0$ (*i.e.*, exponents in $\mathbb{C}$, although $\Rightarrow^{\mathsf{M}}$ does not behave as a categorical exponent).

(A.iv) Abstraction – Currying $\Phi_{A,B} : \mathbb{C}_{\mathsf{M}}(X \times^{\mathsf{M}} A, Y) \to \mathbb{C}_{\mathsf{M}}(X, A \Rightarrow^{\mathsf{M}} Y)$:
- defined $\Phi f = \eta \circ \phi f$ using the underlying adjunction ϕ of $\mathbb{C}$. Or diagrammatically:

$$\Phi_{A,B} := \mathbb{C}(A \times X, \mathsf{M}B) \xrightarrow{\phi_{A,B}} \mathbb{C}(A, X \Rightarrow \mathsf{M}B) \xrightarrow{\mathbb{C}(-,\eta)} \mathbb{C}(A, \mathsf{M}(X \Rightarrow \mathsf{M}B)) \tag{25}$$

(A.v) Application – Uncurrying $\Psi_{A,B} : \mathbb{C}_{\mathsf{M}}(X, A \Rightarrow Y) \to \mathbb{C}_{\mathsf{M}}(X \times A, Y)$:
- defined $\Psi f = \mathsf{ev}^* \circ \mathsf{st}' \circ (f \times id)$ using underlying adjunction ϕ^{-1} of $\mathbb{C}$. Or diagrammatically (using the natural transformation $(- \times id)_{A,B} : (A \Rightarrow B) \to (A \times X \Rightarrow B \times X)$):

$$\Psi_{A,B} := \mathbb{C}(A, \mathsf{M}(X \Rightarrow \mathsf{M}B)) \xrightarrow{(-\times id_X)} \mathbb{C}(A \times X, \mathsf{M}(X \Rightarrow \mathsf{M}B) \times X) \xrightarrow{\mathbb{C}(-,\mathsf{ev}^* \circ \mathsf{st}')} \mathbb{C}(A \times X, \mathsf{M}B) \tag{26}$$

(B.i) [$\equiv$-*let*-λ] – $\Psi \circ \Phi = id$:
- The proof follows from the above definitions of Φ and Ψ for $\mathbb{C}_{\mathsf{M}}$ and from the properties of a strong monad and underlying products and exponents of $\mathbb{C}$:

$$\begin{aligned}
(\Psi \circ \Phi)f &= \Psi(\eta \circ \phi\, f) && \{\Phi \textit{ definition } (25)\}\\
&= \mathsf{ev}^* \circ \mathsf{st}' \circ ((\eta \circ \phi\, f) \times id) && \{\Psi \textit{ definition } (26)\}\\
&= \mathsf{ev}^* \circ \mathsf{st}' \circ (\eta \times id) \circ (\phi f \times id) && \{\textit{functoriality of } \times\}\\
&= \mathsf{ev}^* \circ \eta \circ (\phi f \times id) && \{\textit{strong monad (Appendix B.1.4)}\}\\
&= \mathsf{ev} \circ (\phi f \times id) && [K2]\\
&= f && \{\textit{universal property of } \Rightarrow\} \quad \square
\end{aligned}$$

  Thus [$\equiv$-*let*-λ] holds, *i.e.*, $(\lambda x.e_2)e_1 \equiv$ **let** $x = e_1$ **in** e_2 in Moggi's calculus.

(B.ii) $[\equiv\text{-}\beta]$ – via inductive proof or $[\equiv\text{-}let\text{-}\lambda]$ with an inductive proof of $[let\text{-}\beta]$:

– Consider, for example the following term (in CBV) **let** $x = print$ `"foo"` **in** 0. If $[let\text{-}\beta]$ holds, this is equivalent to the integer term 0. However, the first term has the side-effect of printing `foo` while the second does not. Thus, $[let\text{-}\beta]/[\equiv\text{-}\beta]$ does not hold for λ_c. Instead a restricted form of β-reduction holds for application to variables only, rather than arbitrary expressions (proof omitted here):

$$[\equiv\text{-}\beta\text{-VAR}]\frac{}{\Gamma \vdash (\lambda x.e)v \equiv e[x := v]}$$

(B.iii) $[\equiv\text{-}\eta]$ – $\Phi \circ \Psi = id$ and either (21) or (22):

– Although Ψ is the left-inverse of Φ the inverse composition does not give an identity $\Phi \circ \Psi \neq id$. The proof is straightforward by counterexample with a little calculation.

Proof. Consider the partiality monad $\mathsf{M}A = A+1$. Recall that for the partiality monad $\eta = inl$. Let $f : \Gamma \to \mathsf{M}(\tau \to \mathsf{M}\tau')$ be defined as $f = \lambda\Gamma\,.\,inr\,1$ then:

$$\begin{aligned}(\Phi \circ \Psi)f &= \Phi(\mathsf{ev}^* \circ \mathsf{st}' \circ (f \times id)) \\ &= \eta \circ \phi(\mathsf{ev}^* \circ \mathsf{st}' \circ (f \times id))\end{aligned}$$

No further calculation is needed to see the contradiction. Since the last function in the composition is η, we know that at least:

$$\begin{aligned}(\Phi \circ \Psi)f &= \lambda\Gamma\,.inl\,((\mathsf{ev}^* \circ \mathsf{st}' \circ (f \times id))\,\Gamma) \\ \neq \quad f &= \lambda\Gamma\,.inr\,1\end{aligned}$$

No further calculation is needed to derive the contradiction; $\Phi \circ \Psi \neq id$. □

(B.iv) Symmetric monoidality of $\times^{\mathsf{M}}$ for *contraction*, *weakening*, and *exchange*;

– $\mathbb{C}_{\mathsf{M}}$ is not symmetric monoidal as the $\times^M$ and $\Rightarrow^M$ are not categorical products and exponents. $\mathbb{C}_{\mathsf{M}}$ is *pre-monoidal* (*i.e.*, weaker than a monoidal category) [**PR97**].

Summary. It was shown here that $\mathbb{C}_{\mathsf{M}}$ has hom-set-adjunction-like operations $\Phi_{X,Y}$ and $\Psi_{X,Y}$ which provide the structure to give a semantics to λ_c, but with a weaker base equational theory than λ^τ, with only Ψ as the left-inverse of Φ, *i.e.*, there is no adjunction $(- \times^{\mathsf{M}} -) \dashv (- \Rightarrow^{\mathsf{M}} -)$. Thus, $\mathbb{C}_{\mathsf{M}}$ for a strong monad M is said be *pre Cartesian-closed* [**UV08**], or alternatively a *Freyd category* with Kleisli exponentials (see, *e.g.*, [**PT99**]).

2.4. Categorical programming

Whilst the approach of categorical semantics uses category theory as a metalanguage for programming language semantics, *categorical programming* uses a programming language as a metalanguage for category theory; the roles of categories and programming languages are swapped. The categorical programming approach views programs as categorical definitions and can be roughly split into two (sometimes overlapping) activities: reasoning about programs using categorical laws and organising/abstracting programs using categorical concepts. The former is grounded in the long-standing tradition of *algebraic* approaches to programming.

Algebra of programming. The *algebraic* approach to programming views programs symbolically, where a semantically equivalent program can be calculated by applying equalities on sub-parts of a program akin to standard algebra. Languages based on Church's λ-calculus may inherit the underlying equational theory. For example, Landin's ISWIM includes, amongst others, a β-equality law [**Lan66**]. Landin notes that β is only generally applicable for purely-functional ISWIM programs. Burstall and Landin developed the algebraic approach further to verify a simple compiler [**BL69**]. Backus' well known paper advocated the algebraic approach in functional programming for reasoning and program construction [**Bac78**].

Bird extended the algebraic approach to operations on lists [**Bir87**], in collaboration with Meertens who furthered the algebraic approach [**Mee86**]). The Bird-Meertens approach was extended to other data types (*e.g.*, to trees and arrays [**B$^+$88, Jeu89**]). The algebraic approach was used, not only for reasoning about programs, but also for deriving programs systematically from a specification [**MFP91, G$^+$94, Gib03**].

The algebra-of-programming approach gradually became more influenced by category theoretic concepts further than the laws following from the simply-typed λ-calculus and CCCs. Wadler's *Theorems for free!* showed that algebraic laws could be inferred from the semantics of parametric polymorphism and the types of polymorphic functions [**Wad89**]. Simultaneously, de Bruin showed a similar technique, relating these theorems to the naturality property of natural transformations [**dB89**]. Spivey identified that many of the laws in Bird's *theory of lists* [**Bir87**] correspond exactly to category-theoretic notions of functors, natural transformations, adjunctions, and algebras [**Spi89**]. Initial and final algebras of arbitrary data types were used to capture various recursion schemes [**Mal90, MFP91, FM91**].

These references are not exhaustive, and much work has followed. Crucially, category theory provides both an elegant framework for equational reasoning and a toolkit of general mathematical structures that appear to be extremely applicable to programming. As Fokkinga points out, the language of category theory "often suggests or eases a far-reaching generalisation" [**FOK92**].

Programming idioms. Closely related to the algebraic approach is the concept of using concepts from category theory as tools for abstracting and organising programs. For example, Vene's thesis, fittingly titled *Categorical programming with inductive and coinductive types*, uses categorical concepts both for reasoning about, and intentionally structuring, programs [**Ven00**].

Another example is the use of *monads*. Following Moggi's seminal work on monads for abstracting the semantics of impure language features [**Mog89, Mog91**], Wadler showed that monads can be used as a programming idiom for encoding and abstracting various impure notions of computation in a pure language [**Wad92a, Wad92b, Wad95b**].

There are now many categorical concepts that are used directly in programs as first-class entities, including *functors*, *monads*, *monoidal functors* (*applicative functors*) [**MP08**], *Freyd categories* (under the name *arrows*) [**Hug00, Hug05, Pat01**], and now also comonads [**UV06, UV07, OBM10**]. This section reviews categorical programming with functors and monads in Haskell, as a precursor to categorical programming with comonads in the next chapter.

Idioms apart from laws. The use of category theory as a programming idiom can be seen separately from the algebraic approach since category theoretic concepts may be used in languages which do not have a well-defined categorical semantics (or, if the semantics did exist, it would be baroque). In these languages, the calculational/algebraic approach to programming may be less useful, since the number of exceptions to categorical laws may render them largely misleading. However, the category theoretic concepts still provide a useful set of programming idioms for organising and abstracting programs.

Algebraic reasoning, using categorical laws, may still be useful, even if the laws do not always hold, for example, many laws are violated by *partiality*, either from exceptions or non-termination. Danielsson *et al.* showed that, in a partial language, algebraic reasoning about a finite and total (terminating) subset of programs (*i.e.*, without $\bot$) is "morally correct" since it is impossible to transform terminating programs into non-terminating programs [**DHJG06**]. This was demonstrated for a simple language of a bi-Cartesian closed category (BCCC) with recursive polymorphic types and fold/unfold operations on these, resembling a subset of Haskell. Their results transfer to Haskell, where non-termination is usually unintended in most programs, and thus the axioms of a BCCC may be assumed for the subset finite and total Haskell programs (modulo type classes and other more advanced type system features).

Thus, Haskell is well-suited to categorical programming due to its BCCC-like semantics, coupled with its applicative syntax and powerful abstraction mechanisms. Indeed, a plethora of category theoretic concepts are common in Haskell.

2.4.1. Categorical view of programs

Categorical programming can be guided by matching the *type structure* of programs to the *signatures* of morphisms in category theory (including morphisms between categories, *i.e.*, functors, and between functors, *i.e.*, natural transformations). This can be done by interpreting the types and type structure of a program into some abstract category $\mathbb{P}$. The following informally describes the approach.

Monomorphic types. Monomorphic types are objects of $\mathbb{P}$, and monomorphically-typed functions are morphisms between their parameter and return type objects [**FOK92**], *e.g.*, the function $f : A \to B$ is a morphism $f : A \to B \in \mathbb{P}_1$. Expressions which do not have a function type are interpreted as morphisms from a terminal object in the category $1 \in \mathbb{P}_0$. This interpretation contrasts with the categorical semantics approach where terms (rather than functions) are interpreted as morphisms, with the free-variable context as the source object.

For a language with first-class functions, a *function type* is required. Thus, $\mathbb{P}$ is a closed category with exponent objects $A \Rightarrow B \in \mathbb{P}_0$ for all $A, B \in \mathbb{P}_0$.

Polymorphic types. Many category-theoretic concepts are defined universally over the objects of a category, *e.g.*, "for every object X in $\mathbb{C}[...]$". Since types are understood as objects in $\mathbb{P}$, universal quantification over types provided by the polymorphic λ-calculus is central to categorical programming with concepts beyond simple categories.

For categorical programming in this dissertation, polymorphism is interpreted as a metatheoretic concept, where polymorphic types and polymorphically-typed functions capture *parameterised families* of objects (object mappings) and morphisms respectively. For example, the type $\forall a, b.\tau$ is an object mapping from a pair of types $a \times b \in (\mathbb{P} \times \mathbb{P})_0$ (*i.e.*, objects of the product category $\mathbb{P} \times \mathbb{P}$) to a type $\tau \in P_0$.

The type constructors of parametric data types can be similarly interpreted as object mappings, *e.g.*, a parametric data type F with a single parameter (*e.g.*, in Haskell, **data** $F\ a = ...$) is an object mapping $F : \mathbb{P}_0 \to \mathbb{P}_0$.

Subsequently, various categorical structures can be defined by polymorphic functions between type constructors, defining *structure morphisms*. For example, *projections* for products can be described by natural transformations, *i.e.* $\pi_{1\,A,B} : A \times B \to A$. As a natural transformation, π_1 is a family of morphisms indexed by A, B objects. This can be captured as a polymorphic function, which similarly provides a family of functions parameterised by types:

$$\Lambda a.\Lambda b.\lambda(a, b).a : \forall a, b.(a, b) \to a$$

A similar approach can be used to encode functors.

Functors. Reynolds and Plotkin previously characterised *definable* functors which, for a categorical semantics of the polymorphic-λ calculus in terms of a Cartesian-closed category $\mathbb{C}$, are endofunctors whose morphism mapping can be expressed as a term in the polymorphic-λ calculus [**RP93**]. Indeed, given an object-mapping interpretation of a parametric data type $F : \mathbb{P}_0 \to \mathbb{P}_0$, a morphism-mapping can be defined within $\mathbb{P}$ as a polymorphic function of type:

$$\mathit{fmap} : \forall \alpha\, \beta.(\alpha \Rightarrow \beta) \to (F\, \alpha \Rightarrow F\, \beta)$$

which should satisfy analogous functorial laws: $\mathit{fmap}\ (g \circ f) = \mathit{fmap}\ g \circ \mathit{fmap}\ f$ and $\mathit{fmap}\ \mathit{id} = \mathit{id}$.

This embedding of the morphism mapping corresponds to the notion of a *strong* or *enriched functor*. *Enriched category theory* generalises categories by allowing the *hom-set* (the collection of morphisms between two objects in a category) to be an object of another (*enriching*) category $\mathbb{V}$ [**Kel82**]. Composition is then defined as morphism in $\mathbb{V}$ between these objects. An enriched functor F defines its morphism mapping as an indexed family of morphisms in $\mathbb{V}$ between the $\mathbb{C}(A, B)$ and $\mathbb{C}(\mathsf{F}A, \mathsf{F}B)$ hom-set objects (see Appendix B.1.5 (p. 189) for definitions).

Every closed monoidal category (*e.g.*, CCCs) is *self enriched* where the hom-set $\mathbb{C}(A, B)$ is internalised by the exponential $A \Rightarrow B \in \mathbb{C}_0$ of the function type from A to B. Thus, a definition of *fmap* of the above type describes a self-enriched functor (also called *strong functor* [**Koc72**]). This is equivalent to the notion of *tensorial strength* discussed earlier in the context of strong monads (Section 2.3.1) (see Appendix B.1.4, (p. 188) for more).

Enriched vs. internal. Definitions of category theoretic concepts, such as functors, may be formalised by their *internalisation* within an *ambient category* $\mathbb{A}$. Rather than describing a functor informally as having an object and morphism mapping, a functor is described by morphisms $F_0 : \mathbb{C}_0 \to \mathbb{D}_0 \in \mathbb{A}_1$ for object mapping and $F_1 : \mathbb{C}_1 \to \mathbb{D}_1 \in \mathbb{A}_1$ for morphism mapping.

Enrichment and *internalisation* provide two ways of generalising category-theoretic concepts (for which there are ways to interchange between the approaches [**Ver92**]). The approach here uses *enrichment*, rather than internalising structure within an ambient category, since $\mathbb{P}$ is not equipped with objects of all types and morphisms and does not have F_0 and F_1 morphisms.

2.4.2. Categorical programming in Haskell

The above categorical programming interpretation of the polymorphic λ-calculus applies to the total (finite) subset of Haskell programs (without features like type classes). Throughout this dissertation, the abstract category of (total) Haskell programs is referred to as **Hask**, where objects are Haskell types and morphisms are Haskell functions. This is not however the domain of a categorical semantics for full Haskell. It merely represents a space in which Haskell programs are treated as categorical definitions for the sake of categorical programming (structuring programs using categorical concepts). Alternatively, **Hask** may be thought of as a categorical analogy for Haskell programs.

Here we briefly summarise two common categorical programming techniques in Haskell: functors and monads. These are traditionally described via type classes (briefly introduced in Appendix D.1), associating a type constructor (analogous to an object mapping) with a number of functions involving the constructor (analogous to natural transformations).

Functors. For example, the following class in Haskell describes functors:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

(Haskell's typing syntax uses a double colon rather than the standard single colon in type theory).

Thus, by the interpretation in the previous section, instances of *Functor* describe *strong functors*, comprising an object mapping $f : \mathbf{Hask}_0 \to \mathbf{Hask}_0$ and a family of morphisms $(a \Rightarrow b) \to (fa \Rightarrow fb) \in \mathbf{Hask}_1$ (equivalent to mappings from $a \to b \in \mathbf{Hask}_1$ to $fa \to fb \in \mathbf{Hask}_1$).

The prototypical example of a *Functor* is the list data type with the standard *map* operation:

```
instance Functor [] where fmap = map
```

Thus functors are explicit first-class entities in a program, constructed from general language features, rather than a hidden semantic device. Whilst Haskell encourages this kind of categorical structure, it does not enforce any of the categorical laws, which should be checked by hand.

Note that *Functor* is a slight misnomer: only *endofunctors* on **Hask** are captured by its instances. This restriction is discussed later in Section 5.1.

Monads. Monads in Haskell are traditionally modelled in Kleisli triple form by the class:

```
class Monad m where
  return :: a → m a                  -- unit
  (>>=) :: m a → (a → m b) → m b     -- extension
```

over an object mapping $m : \mathbf{Hask}_0 \to \mathbf{Hask}_0$, which should satisfy analogous laws to [K1-3] shown in Section 2.3. The unit natural transformation is provided by *return*, and an infix version of *extension* is provided by $(\gg\!=)$ (pronounced *bind*). Note that *bind* here is self enriched (strong), in the sense that it is embedded as a function.

Haskell provides a syntax to simplify programming with monads, called the **do**-notation which provides a kind of *let*-binding of the form $\mathbf{do}\ x \leftarrow e1; e2$ (where $e1 :: m\ a$, $e2 :: m\ b$, $x :: a$, and the whole expression has type $m\ b$). The expression *e2* may be a final expression for the block or further bindings (see [**PJ+03**, Section 3.14]). The notation is desugared into the operations of a monad (discussed in Chapter 4), essentially following the semantics of *let*-binding in Moggi's λ^c, where:

$$[\![\mathbf{do}\ x \leftarrow e_1; e_2]\!] = e_1 \gg\!= (\lambda x \to [\![e_2]\!]) \tag{27}$$

2.4.3. Summary

The categorical programming approach provides a view of programs as definitions within some category $\mathbb{P}$. For a functional language, with first-class functions, this category is closed (self enriched). A language with polymorphism quantifies over types, allowing various category theory concepts to be encoded (perhaps in enriched forms, such as functors here).

A categorical-programming approach may use categorical concepts as design patterns to aid the organisation and abstraction of programs. A language may provide additional syntax parameterised by a particularly categorical concept, for example: **do**-notation for monads in Haskell [**PJ+03**], *let!* notation in F# [**PS12**], *computation expressions* for additive monads and other semi-group and monoidal structures in F# [**PS12**], *do-case* notation for *joinads* [**PMS11**, **PS11**], and *arrow notation* for *Freyd categories* in Haskell [**Hug05**, **Pat01**]. Thus categorical structuring of programs further helps to simplify their reading and writing. Chapter 4 introduces a notation for programming with comonads, called the **codo**-notation.

2.5. Conclusion

This chapter introduced categorical semantics and categorical programming. In categorical semantics, well-typed terms are mapped to the morphisms of a category. In categorical programming, category theory concepts are used to structure programs, or, from a different perspective, programs can be understood as defining (or making informal analogies to) category theory notions in some abstract category. The categorical semantics approach was shown for the simply-typed λ-calculus, decomposed into the additional structure required of a category to provide its semantics, modulo any equational theory, and the additional requirements to provide $\beta\eta$-equality. This approach was briefly applied to Moggi's *computational* calculus and its categorical semantics for effectful computations using strong monads.

The next chapter applies the schema provided in this chapter to explore a categorical semantics for contextual computations using comonads. Categorical programming with comonads is also discussed, along the same lines as the approach described in Section 2.4 here.

Chapter 3

Comonads

The previous chapter introduced the approach of using category theory to organise and abstract semantics and programs. In the literature, the category theory structure of a *comonad* has been used to describe various notions of computation, often characterised as *context-dependent* computations [**UV08**]. The present chapter reviews and extends existing work using comonads to structure computation. The category theoretic approaches of the previous chapter are applied, developing the semantics and programming of contextual computations using comonads.

Chapter structure and contributions. Section 3.1 reviews the categorical definition of comonads and the standard approach to categorical programming with comonads in Haskell. This chapter then makes the following contributions:

- Sections 3.2 demonstrate various example comonads and constructions on comonads. Examples include various forms of list and tree comonads involving *cursors* and *zippers*.
- A λ-calculus of contextual computations is defined with a categorical semantics in terms of a comonad with additional monoidal structure (Section 3.3), following the systematic approach to categorical semantics of the simply-typed λ-calculus in Chapter 2. This redevelops Uustalu and Vene's semantics for a context-dependent λ-calculus [**UV08**], generalising their semantics and making precise the role of additional structure to provide β- and η-equality to the calculus which requires variations on *monoidal comonads* and *monoidal functors*.
- The role of *shape* in comonadic semantics is studied in Section 3.4. Notably, comonads are shown to be *shape preserving*, meaning that context is propagated uniformally throughout a computation, elucidating the limitations of comonads. Chapter 5 generalises comonads to *indexed comonads*, a consequence of which is that shape preservation is relaxed.
- Section 3.5 compares comonadic and monadic approaches, including a comparison of Uustalu and Vene's comonadic semantics for the dataflow language Lucid [**UV06**] with Wadge's monadic semantics for the language [**Wad95a**].

Brief overview of relevant literature. An early use of comonads in semantics is by Brookes and Geva, describing *intensional semantics* of programs to capture higher-level properties such as evaluation order [**BG91**]. Brookes and Stone extended this work, combining comonads with monads using a *distributive law* [**BS93**] (this approach is used later in Section 5.1.3). Harmer *et al.* similarly combined monads and comonads in the setting of game semantics [**HHM07**].

Kieburtz provided one of the earliest accounts of comonads in programming [**Kie99**]. Uustalu and Vene showed that various notions of contextual computation are captured by comonads, including the semantics of Lucid [**UV05, UV06**] and *attribute grammars* over trees [**UV07**]. They later developed a general calculus of context-dependent computations with a categorical semantics in terms of *monoidal comonads* [**UV08**]. Orchard and Mycroft structured array computations for parallel programming using comonads [**OBM10**], with a similar use by Havel

and Herout describing rendering pipelines [**HH10**]. Array comonads have also been previously commented on by Piponi [**Pip09**]. Related to comonadic arrays, Capobianco and Uustalu described cellular automata comonadically, proving a number of well-known theorems about cellular automata [**CU10**]. Ahman, Chapman, and Uustalu characterised *containers* which have a comonad structure as *directed containers* [**ACU12**].

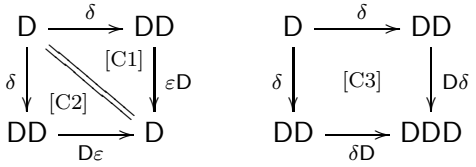
Bierman and de Paiva showed that (monoidal) comonads provide a categorical model of intuitionistic S4 modal logic [**BdP96**, **BdP00**]. The correspondence between comonads and modal logic is outside this dissertation's scope, but is discussed briefly in Section 8.2.3. Various works have used comonads to generalise and capture various recursion schemes on data types, *e.g.*, [**Par00**, **UVP01**, **CUV06**]. This use of comonads is not explored in this dissertation.

3.1. Definitions

Comonads provide operations for the composition of functions/morphisms with *structured input*, of type $\mathsf{D}\,\sigma \to \tau$. Thus, comonads are dual to monads (Section 2.3) which provide composition for morphisms with *structured output*, $\sigma \to \mathsf{M}\,\tau$. Monads and comonads therefore encapsulate *impure* features of a computation; monads capture impurity related to the output (changing the context), traditionally called *side-effects*, and comonads capture impurity related to the input (making demands on the context).

Definition 3.1.1. A *comonad* is a triple $(\mathsf{D}, \varepsilon, \delta)$ of an endofunctor $\mathsf{D} : \mathbb{C} \to \mathbb{C}$ and two natural transformations: $\varepsilon : \mathsf{D} \dot{\to} 1_{\mathbb{C}}$ and $\delta : \mathsf{D} \dot{\to} \mathsf{DD}$ called *counit* and *comultiplication* respectively, satisfying the following axioms [C1-3]:

$$
\begin{array}{ll}
[\text{C1}] & \varepsilon\mathsf{D} \circ \delta = 1_{\mathsf{D}} \\
[\text{C2}] & \mathsf{D}\varepsilon \circ \delta = 1_{\mathsf{D}} \\
[\text{C3}] & \delta\mathsf{D} \circ \delta = \mathsf{D}\delta \circ \delta
\end{array}
$$



From the perspective of contextual computations and objects as types, $\mathsf{D}\,\tau$ encodes computations of pure values of type τ that depend on some notion of context. Alternatively, $\mathsf{D}\,\tau$ is an *intension*, capturing τ-typed values of a computation in all contexts. The morphism mapping $\mathsf{D}\,f : \mathsf{D}X \to \mathsf{D}Y$ enumerates every possible *contextual position* in an intension, which can hold an X value, applying f to each value. The counit operation defines the notion of *current* context, returning the value of the computation at this context. Comultiplication defines accessibility between contexts, producing a contextual computation of inner contextual computations that are "focused", or *localised*, to the context of the outer computation at each position.

This chapter provides various examples (Section 3.2).

Note that the term *context* has three main usages in this dissertation: (1) the informal notion of *context* of execution, giving a program meaning; (2) a value of type $\mathsf{D}A$ (encoding the context of a computation); (3) contextual positions in $\mathsf{D}A$, *e.g.*, *extension* applies a local operation at

all *contexts*. When it is important to distinguish, usage (2) is referred to as the *intension*. For usage (3), *contextual position* is usually used instead of *context* for clarity.

Example 3.1.2. One of the simplest comonads is the *product comonad*, comprising the endofunctor $(- \times P) : \mathbb{C} \to \mathbb{C}$ for an implicit parameter of type P, where $(- \times P)(f : X \to Y) = f \times id_P : X \times P \to Y \times P$, with the comonad operations defined as follows:

- $\varepsilon\,(x, p) = x$ (discard the implicit parameter)
- $\delta\,(x, p) = ((x, p), p)$ (duplicate the implicit parameter)

The product comonad abstracts a simple notion of context in computations: (immutable) implicit parameters of a computation, such as a global variables. Thus enumeration of contexts, and definition of the current context is trivial since there is only a single contextual position.

Lewis *et al.* briefly mention using comonads to structure the semantics of implicit parameters in Haskell [**LLMS00**]. Section 6.5 (p. 135) that the semantics of implicit parameters requires a more general structure than the product comonad.

Example 3.1.3. The functor $[A] = \mu X.A + A \times X$ of non-empty lists, with the usual *map* operation on lists for the morphism mapping, is a comonad with definitions (in Haskell-style notation):

- $\varepsilon\ xs = head\ xs$ (return the head element)
- $\delta\ xs = \begin{cases} [[x]] & xs = [x] \\ xs : \delta(tail\ xs) & otherwise \end{cases}$ (suffix lists rooted at the current position)

For lists, the morphism mapping applies its parameter to each element in the list (the standard *map* operation), enumerating the possible contextual positions. The comultiplication operation *localises* its parameter list at each context by taking the suffix list from the corresponding context, *e.g.*, $\delta\,[1, 2, 3] = [[1, 2, 3], [2, 3], [3]]$. Thus, in each inner list, the element at the current context is the head of the list, as defined by ε. The *non-empty* pre-condition is needed since ε is undefined for empty lists.

The operations of a comonad provide associative and unital composition for morphisms $\mathsf{D}\sigma \to \tau$, defining a *coKleisli category*:[1]

Definition 3.1.4. Given a comonad D on a category $\mathbb{C}$, the *coKleisli category* $\mathbb{C}_{\mathsf{D}}$, has objects $\mathbb{C}_{\mathsf{D}0} = \mathbb{C}_0$ and morphisms $\mathbb{C}_{\mathsf{D}}(X, Y) = \mathbb{C}(\mathsf{D}X, Y)$, for all objects $X, Y \in \mathbb{C}_0$, with:

- *composition*: $g \mathbin{\hat{\circ}} f = \mathsf{D}X \xrightarrow{\delta} \mathsf{D}(\mathsf{D}X) \xrightarrow{\mathsf{D}f} \mathsf{D}Y \xrightarrow{g} Z$ (for all $f : \mathsf{D}X \to Y, g : \mathsf{D}Y \to Z \in \mathbb{C}_1$).
- *identities*: $\hat{id}_X = \mathsf{D}X \xrightarrow{\varepsilon_X} X$ (for all $X \in \mathbb{C}_0$).

The categorical laws of associativity and unitality follow directly from the comonad laws [C1-3].

Example 3.1.5. For the *product comonad*, coKleisli composition duplicates the incoming implicit parameter, passing it to the composed morphisms, *i.e.*, for $f : X \times P \to Y, g : Y \times P \to Z$:

$$(x, p) \underset{\delta}{\longmapsto} ((x, p), p) \underset{\mathsf{D}f}{\longmapsto} (f(x, p), p) \underset{g}{\longmapsto} g(f(x, p), p) \tag{28}$$

[1] The terminology of a *coKleisli category* is used here, appearing in some papers *e.g.* by Uustalu and Vene [**UV06, UV08**], while in mathematical texts the terminology of a *Kleisli category for a comonad* is often used.

Morphisms $\mathsf{D}\sigma \to \tau$ for a comonad D are commonly referred to as *coKleisli morphisms*. From the contextual perspective, coKleisli morphisms compute a value at a context provided by the parameter structure. CoKleisli morphisms are sometimes called *local operations* here, reflecting the idea that the return result is "local" to the context provide by the parameter.

The introductory chapter informally introduced comonads with an *extension* operation which promotes local operations to *global operations*. This operation is derivable from the comonad operations and is provided explicitly by an alternate, but equivalent, presentation of comonads:

Definition 3.1.6. The *coKleisli triple form* of a comonad comprises an object mapping $\mathsf{D} : \mathbb{C}_0 \to \mathbb{C}_0$, an *extension* operation[2] $(-)^\dagger : \mathbb{C}(\mathsf{D}X, Y) \to \mathbb{C}(\mathsf{D}X, \mathsf{D}Y)$, natural in X, Y, and counit operation $\varepsilon_X : \mathsf{D}X \not\to X$, such that for all $f : \mathsf{D}X \to Y$, $g : \mathsf{D}Y \to Z$:

$$\text{[cK1]} \quad (\varepsilon_X)^\dagger = id_{\mathsf{D}X} \qquad \text{[cK2]} \quad \varepsilon_Y \circ f^\dagger = f \qquad \text{[cK3]} \quad (g \circ f^\dagger)^\dagger = g^\dagger \circ f^\dagger$$

Example 3.1.7. For the product comonad, extension is defined: $f^\dagger(x, p) = (f(x, p), p)$

The coKleisli triple form replaces comultiplication and the functor D with *extension* and just an object mapping D. The two presentations are related by the following lemma:

Lemma 3.1.8. *The coKleisli triple and comonoidal (in terms of δ) forms of comonad are equivalent by the equalities:*

$$f^\dagger = \mathsf{D}f \circ \delta \qquad \delta = id_{\mathsf{D}}{}^\dagger \qquad \mathsf{D}f = (f \circ \varepsilon)^\dagger \tag{29}$$

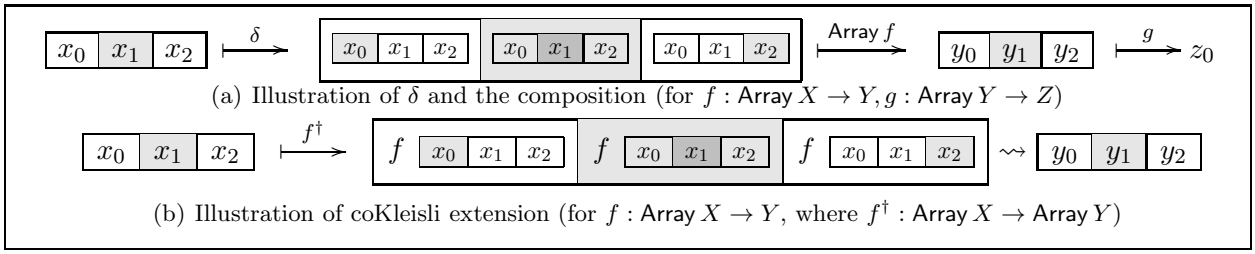
The proof is elided for brevity. By the left-most equation of (29), composition for a coKleisli category can therefore be equivalently defined: $g \mathbin{\hat{\circ}} f = g \circ f^\dagger$.

From the contextual perspective, extension takes a local operation defined at a particular context and promotes, or *extends*, it to a global operation over all contexts. This behaviour can be seen clearly with the array comonad which was informally illustrated in Section 1.2.2. The array comonad is defined more formally in Haskell shortly.

Example 3.1.9. From the contextual perspective, arrays define a value which is dependent on its position (index) in the array. For example, arrays might be used to discretise a continuous variable such as the temperature in a room, which depends on the measurement position (and/or time). The array comonad has a *cursor* pointing to a particular index in the array.

Figure 3.1(a) illustrates comultiplication $\delta : \mathsf{Array}\, X \to \mathsf{Array}(\mathsf{Array}\, X)$ and coKleisli composition. For comultiplication, each position in the outer array has a copy of the original array with its cursor set to its position in the outer. Since the original array is copied to the inner arrays, all array positions are accessible from any other position. **Figure** 3.1(b) illustrates lifting a local operation on arrays to a global operation using extension. The equivalence $f^\dagger = \mathsf{D}f \circ \delta$ can be seen in the diagrams. Example 3.1.10 below formally defines these operations.

[2]Sometimes this operation is called *coextension* since the dual operation for a monad is called the *extension*. This dissertation uses *extension* when it is clear that the comonadic form is meant; otherwise *coextension* is used.



(a) Illustration of δ and the composition (for $f : \mathsf{Array}\, X \to Y, g : \mathsf{Array}\, Y \to Z$)

(b) Illustration of coKleisli extension (for $f : \mathsf{Array}\, X \to Y$, where $f^\dagger : \mathsf{Array}\, X \to \mathsf{Array}\, Y$)

Figure 3.1. Illustration of δ, composition, and extension for the pointed array comonad

3.1.1. Categorical programming

Comonads in Haskell can be described by the following type class:

```
class Functor c ⇒ Comonad c where
  current :: c a → a                    -- counit
  extend :: (c a → b) → c a → c b       -- extension
  extend f = fmap f ∘ disject
  disject :: c a → c (c a)              -- comultiplication
  disject = extend id
```

Thus a comonad in Haskell is analogous to an endofunctor $c : \mathbf{Hask} \to \mathbf{Hask}$, and *current*, *extend*, and *disject* are analogous to the *counit*, *extension*, and *comultiplication* operations respectively. The mutually defined default implementations of *disject* and *extend* require that an instance of *Comonad* defines at least one of the two operations. Unfortunately, Haskell is too weak to express that only an object mapping $c : \mathbf{Hask}_0 \to \mathbf{Hask}_0$ is required if *extend* is defined, rather than a full functor with an instance of *Functor*. In this dissertation, a *Functor* instance is elided if *extend* is defined.

A word on naming. At the time of writing, the *Comonad* class in Haskell is provided by the `Control.Comonad` package, which uses different names for its operations than here [**Kme12**]. The counit is called *extract*. Here *current* is used to avoid confusion between *extract* and *extend*, and to associate the operation with the contextual understanding that it returns the value at the current context. The comultiplication is called *duplicate*. Here *disject* (meaning to scatter, break apart, or distribute) is used instead to give a more general intuition of the operation's behaviour. For the product comonad, *duplicate* is descriptive, but for the non-empty list comonad above, the operation resembles a deconstruction and distribution of the context, rather than a duplication.

Example 3.1.10. The *pointed array comonad* may be defined in Haskell:

```
data CArray i a = CA (Array i a) i

instance Ix i ⇒ Comonad (CArray i) where
  current (CA a i) = a ! i
```

(a) *Lab* image (b) *extend gauss2D* applied to *Lab*

Figure 3.2. Example of a local operator applied to a *CArray* comonad for an image.

$$\begin{aligned} \mathit{extend}\ f\ (\mathit{CA}\ a\ i) = \mathbf{let}\ \mathit{es}' &= \mathit{map}\ (\lambda j \to (j, f\ (\mathit{CA}\ a\ j)))\ (\mathit{indices}\ a) \\ \mathbf{in}\ \ \mathit{CA}\ &(\mathit{array}\ (\mathit{bounds}\ a)\ \mathit{es}')\ i \end{aligned}$$

where *CArray* pairs Haskell's *boxed* array type with a particular index of the array, *current* accesses the cursor element using the array indexing operation !, and, for every index j of the parameter array, *extend* applies f to the array with j as its cursor, returning an index-value pair list from which the result array is constructed. Various operations of *Array* type are used:

$\mathit{bounds} :: \mathit{Ix}\ i \Rightarrow \mathit{Array}\ i\ e \to (i, i)$ -- computes the bounds of an array
$\mathit{array} :: \mathit{Ix}\ i \Rightarrow (i, i) \to [(i, e)] \to \mathit{Array}\ i\ e$ -- constructs an array
$\mathit{indices} :: \mathit{Ix}\ i \Rightarrow \mathit{Array}\ i\ e \to [i]$ -- returns a list of the indices of an array
$(!) :: \mathit{Ix}\ i \Rightarrow \mathit{Array}\ i\ e \to i \to e$ -- indexes an array

Note, that the parameter array and return array in *extend* have the same size and cursor, *i.e.*, *extend* preserves the incoming context's shape in its result.

Many array operations can be defined as local operations using relative indexing, *e.g.*, *convolution operators* common in image and signal processing, such as the following discrete *Laplacian* and *Gaussian* operators for two-dimensional arrays:

$$\begin{aligned} &\mathit{gauss2D}, \mathit{laplace2D} :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float} \to \mathit{Float} \\ &\mathit{gauss2D}\ a = (a\ ?\ (-1, 0) + a\ ?\ (1, 0) + a\ ?\ (0, -1) + a\ ?\ (0, 1) + 4 * a\ ?\ (0, 0))\ /\ 6 \\ &\mathit{laplace2D}\ a = a\ ?\ (-1, 0) + a\ ?\ (1, 0) + a\ ?\ (0, -1) + a\ ?\ (0, 1) - 4 * a\ ?\ (0, 0) \end{aligned}$$

where (?) abstracts relative indexing with bounds checking and default values:

$$\begin{aligned} &(?) :: (\mathit{Ix}\ i, \mathit{Num}\ a, \mathit{Num}\ i) \Rightarrow \mathit{CArray}\ i\ a \to i \to a \\ &(\mathit{CA}\ a\ i)\ ?\ i' = \mathbf{if}\ (\mathit{inRange}\ (\mathit{bounds}\ a)\ (i + i'))\ \mathbf{then}\ a\ !\ (i + i')\ \mathbf{else}\ 0 \end{aligned}$$

Section 7.1.4 (p. 147) discusses alternate methods for abstracting boundary checking and values.

Whilst *gauss2D* computes the Gaussian operator at a single context (locally), *extend gauss2D* computes the Gaussian at every context (globally), returning the resulting image. **Figure** 3.2 demonstrates the output of applying *gauss2D* using *extend* to an image stored in an array .

3.2. Examples

The following section provides example comonads. Some are defined in Haskell, where for brevity some definitions will not include the full **instance** for the *Comonad* class.

3.2.1. Products and coproducts

The *product comonad* with functor $\mathsf{D}X = X \times P$, where P is some fixed object was shown in Example 3.1.2. Accompanying Haskell code is included for in Appendix E.2.1 (p. 222).

Alternatively, products provide a comonad of non-empty fixed-length lists, *e.g.*, two-element lists have the functor $\mathsf{D}X = X \times X$ where $\mathsf{D}f = f \times f$. The trivial *identity comonad*, with functor $\mathsf{Id}\,X = X$ and all its operations as the identity, is a special case of a fixed-length list of length one. For a fixed-length list comonad, a particular element position must be chosen as the current context. Enumeration of all possible contexts is provided by permuting the list data through this position. For example, a length-two list comonad may take the first position as the current, *i.e.*, counit $\varepsilon(x, y) = x$ and comultiplication $\delta(x, y) = ((x, y), (y, x))$. Alternatively, the second position can be the current where $\varepsilon(x, y) = y$, thus the permutation of data in comultiplication must be flipped to satisfy the comonad laws, $\delta(x, y) = ((y, x), (x, y))$.

For lists of length greater than two, the comonad laws imply that the permutation can only be a *left-rotation*, and not some other arbitrary permutation. Thus, $\varepsilon(x, y, z) = x$ and $\delta(x, y, z) = ((x, y, z), (y, z, x), (z, x, y))$.

Coproducts. Whilst the functor $\mathsf{F}A = A{+}1$ is a monad, it is not a comonad since $\varepsilon_A : A{+}1 \to A$ is not *total*; it is undefined for $\varepsilon\ (\mathsf{inr}\,1)$. The functor $\mathsf{F}A = A{+}A$ is however a comonad, equivalent to the product comonad $\mathsf{F}A = A{\times}2$ where 2 is a two-valued type. In general, given two comonads F, G then the functor $\mathsf{D}A = \mathsf{F}A + \mathsf{G}A$ is a comonad (code in Appendix E.2.2 (p. 222)).

3.2.2. Trees

Labelled binary trees are an example of a recursively defined data type which is a comonad, and for which there are a number of related comonads involving labelled binary trees with extra structure. The following definition of labelled trees in Haskell is used here:

```
data Tree a = Leaf a | Node a (Tree a) (Tree a)

instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Node x l r) = Node (f x) (fmap f l) (fmap f r)
```

3.2.2.1. Suffix trees

The following *Tree* comonad defines the root node as the current position:

```
instance Comonad Tree where
  current (Leaf x) = x
  current (Node x _ _) = x
```

$$disject\ (Leaf\ x) = Leaf\ (Leaf\ x)$$
$$disject\ (Node\ x\ l\ r) = Node\ (Node\ x\ l\ r)\ (disject\ l)\ (disject\ r)$$

Comultiplication recursively decomposes its parameter tree computing suffix trees, where each position in the tree becomes the cursor (the root), as depicted in **Figure** 3.3(a). The derived extension operation thus applies a local operation $k : Tree\ a \to b$ to each suffix tree. Local operations may only access the current position or those after it (the children).

3.2.2.2. Pointed trees

Comonads with a *pointer* or *cursor*, pair a data type with an address-like value marking the position of the current context. For binary tress, a suitable pointer can be provided by a sequence of booleans, thus a *pointed* labelled binary tree is defined:

$$\textbf{data}\ PTree\ a = PTree\ (Tree\ a)\ [Bool]$$

where *False* and *True* denote the left-branch and right-branch respectively. The cursor therefore points to the current position as follows:

$$\begin{aligned}
&\textbf{instance}\ Comonad\ PTree\ \textbf{where}\\
&\quad current\ (PTree\ (Leaf\ x)\ [\,])\ \ \ \ \ \ = x\\
&\quad current\ (PTree\ (Node\ x\ _\ _)\ [\,]) = x\\
&\quad current\ (PTree\ (Node\ _\ l\ r)\ (x : xs)) = \textbf{if}\ x\ \textbf{then}\ current\ (PTree\ r\ xs)\ \textbf{else}\ (PTree\ l\ xs)
\end{aligned}$$

The comonad laws imply that comultiplication must associate to every contextual position a copy of the original tree along with a (unique) cursor corresponding to that contextual position. The *current* operation deconstructs the cursor to select the current tree, which by the comonad laws is inverse to comultiplication's construction of a cursor. Comultiplication is formally defined:

$$\begin{aligned}
disject\ (PTree\ x\ c) = \textbf{let}\ &disject'\ (Leaf\ _)\ c' = PTree\ (Leaf\ (PTree\ x\ c'))\ c'\\
&disject'\ (Node\ _\ l\ r)\ c' = \textbf{let}\ (PTree\ l'\ _) = disject'\ l\ (c' +\!\!+ [False])\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (PTree\ r'\ _) = disject'\ r\ (c' +\!\!+ [True])\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{in}\ PTree\ (Node\ (PTree\ x\ c')\ l'\ r')\ c'\\
&\qquad\qquad\qquad (PTree\ x'\ _) = disject'\ x\ [\,]\\
&\quad \textbf{in}\ PTree\ x'\ c
\end{aligned}$$

Thus, each label of the returned tree has the parameter tree x paired with the cursor for that context. Therefore, at every contextual position, every other contextual position is accessible. This contrasts with the suffix tree comonad where only the child positions are accessible. **Figure** 3.3(b) illustrates the above definition.

Section 8.2.3 (p. 170) discusses accessibility further, considering a connection between δ and *accessibility relations* in Kripke semantics for modal logic.

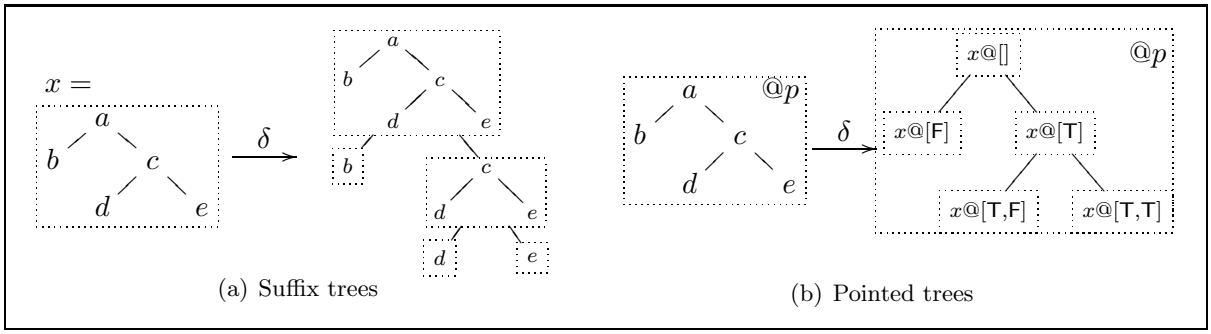


Figure 3.3. Comultiplication for two different labelled binary tree comonads.

3.2.2.3. Tree zipper

Huet's *zipper* data type defines a representation of trees decomposed into a subtree that is the "*focus of attention*" and the remaining parts of the tree not under focus [**Hue97**]. The tree-zipper data type provides a kind of *structural* cursor for trees, where the current context is the subtree in focus. Thus, the zipper captures a notion of context for trees (sometimes called the *type of one-hole contexts* [**McB01**]), allowing operations to be defined locally on tree positions.

Huet describes the zipper for leaf-labelled rose trees; the presentation here instead uses node-and-leaf labelled trees similarly to the tree zipper comonad previously described by Uustalu and Vene [**UV07**]. The zipper type comprises a pair of a tree currently in focus and a *path* which marks the position of the focus tree in the wider context of the parent tree.

$$\textbf{data}\ \mathit{ZTree}\ a = \mathit{TZ}\ (\mathit{Path}\ a)\ (\mathit{Tree}\ a)$$

$$\textbf{data}\ \mathit{Path}\ a = \mathit{Top} \mid \mathit{L}\ a\ (\mathit{Path}\ a)\ (\mathit{Tree}\ a) \mid \mathit{R}\ a\ (\mathit{Path}\ a)\ (\mathit{Tree}\ a)$$

The path is a sequence of direction makers, *left* or *right*, providing the address of the focus tree starting from the root, where each marker is accompanied with the label of the parent node and the subtree of the branch not-taken, *i.e.*, a path going *left* is paired with the *right* subtree, which is not on the path to the focus tree.

Navigation operations move the focus around a tree, often described as operations that "zip" and "unzip" a tree. A tree can be "zipped upwards" by unpacking the head of a path, constructing a new subtree that encompasses the parent node and its non-taken branch, defined:

$$\begin{aligned}
&\mathit{up} :: \mathit{ZTree}\ a \to \mathit{ZTree}\ a \\
&\mathit{up}\ (\mathit{TZ}\ \mathit{Top}\ t) = \mathit{TZ}\ \mathit{Top}\ t && \text{-- already at the top of the tree} \\
&\mathit{up}\ (\mathit{TZ}\ (\mathit{L}\ x\ p\ r)\ t) = \mathit{TZ}\ p\ (\mathit{Node}\ x\ t\ r) && \text{-- zip up from a left branch} \\
&\mathit{up}\ (\mathit{TZ}\ (\mathit{R}\ x\ p\ l)\ t) = \mathit{TZ}\ p\ (\mathit{Node}\ x\ l\ t) && \text{-- zip up from a right branch}
\end{aligned}$$

The following two operations, $\mathit{left}, \mathit{right} :: \mathit{ZTree}\ a \to \mathit{ZTree}\ a$, unzip "downwards", taking either the left or right branch of the focus tree and extending the path accordingly:

$$\begin{aligned}
&\mathit{left}\ (\mathit{TZ}\ p\ (\mathit{Leaf}\ x)) = \mathit{TZ}\ p\ (\mathit{Leaf}\ x) && \text{-- at a leaf} \\
&\mathit{left}\ (\mathit{TZ}\ p\ (\mathit{Node}\ x\ l\ r)) = \mathit{TZ}\ (\mathit{L}\ x\ p\ r)\ l && \text{-- focus on left subtree, add right to path}
\end{aligned}$$

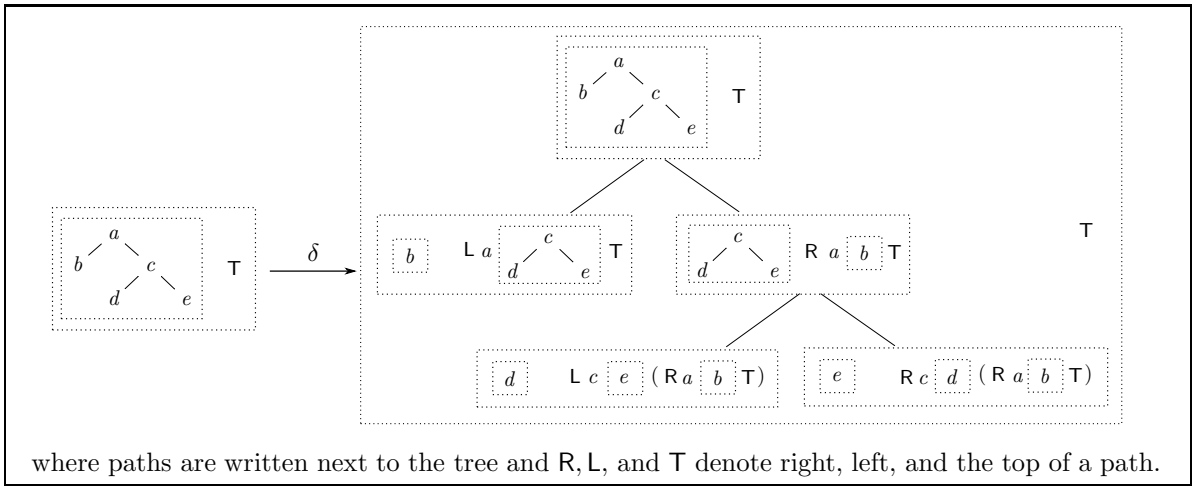


Figure 3.4. Example comultiplication for binary tree zipper comonad

$right\ (TZ\ p\ (Leaf\ x)) = TZ\ p\ (Leaf\ x)$ -- at a leaf
$right\ (TZ\ p\ (Node\ x\ l\ r)) = TZ\ (R\ x\ p\ l)\ r$ -- focus on right subtree, add left to path

These navigation operations move towards a limit of a tree (either towards the root for *up*, or the leaves for *left* and *right*) where reaching a limit of a tree does not cause an error— the navigation operations remain at the limit. Thus, the following coherence conditions are satisfied:

$$(up \circ left)\ x = x \qquad \text{if } left\ x \neq x$$
$$(up \circ right)\ x = x \qquad \text{if } right\ x \neq x$$

Alternatively, partial navigation operations of type $ZTree\ a \to Maybe\ (ZTree\ a)$ may be defined where navigating beyond the limits returns *Nothing*. Various other primitive were provided by Huet for insertions and deletions from the tree zipper, which are omitted here for brevity.

The tree zipper is a comonad with the label of the tree in focus as its cursor:

instance *Comonad ZTree* **where**
$current\ (TZ\ p\ (Leaf\ x)) = x$
$current\ (TZ\ p\ (Node\ x\ \_\ \_)) = x$

Comultiplication refocusses the zipper at each context. **Figure** 3.4 illustrates its operation for a tree which is currently focussed at its root, *i.e.*, the path is *Top*. The operation is defined as follows (based on the definition given by Uustalu and Vene [**UV07**]):

$disject\ tz = TZ\ (disjectP\ tz)\ (disjectT\ tz)$
$disjectT :: ZTree\ t \to Tree\ (ZTree\ t)$ -- comultiplication of tree in focus
$disjectT\ t@(TZ\ p\ (Leaf\ x)) = Leaf\ t$
$disjectT\ t = Node\ t\ (disjectT \circ left\ \$\ t)\ (disjectT \circ right\ \$\ t)$
$disjectP :: ZTree\ t \to Path\ (ZTree\ t)$ -- comultiplication of path

$$
\begin{aligned}
&\mathit{disjectP}\ (\mathit{TZ}\ \mathit{Top}\ t) = \mathit{Top}\\
&\mathit{disjectP}\ z@(\mathit{TZ}\ (L\ \_\ \_\ \_)\ \_) = L\ (\mathit{up}\ z)\ (\mathit{disjectP}\ (\mathit{up}\ z))\ (\mathit{disjectT}\ (\mathit{right} \circ \mathit{up}\ \$\ z))\\
&\mathit{disjectP}\ z@(\mathit{TZ}\ (R\ \_\ \_\ \_)\ \_) = R\ (\mathit{up}\ z)\ (\mathit{disjectP}\ (\mathit{up}\ z))\ (\mathit{disjectT}\ (\mathit{left} \circ \mathit{up}\ \$\ z))
\end{aligned}
$$

General zippers. McBride showed the remarkable technique of computing a zipper, or type of one-hole contexts, for a polynomial data type by applying *partial differentiation* to the algebraic structure of the data type, with respect to the parameter type variable [**McB01**, **AAMG04**]. Given a (regular) data type $\mathsf{D}A$, its zipper comonad is then defined as the data type $\mathsf{D}'A = A \times \partial_A(\mathsf{D}A)$, pairing a zipper with the element at the cursor position. The usual partial differentiation rules apply, interpreting sum type as addition, product types as multiply, function spaces as exponents, and constants as finite types. Fixed points are differentiated by the rule:

$$\partial_A(\mu X.F) = \mu Z.(\partial_A F)_{[X \mapsto \mu \mathsf{X.F}]} + (\partial_X F)_{[X \mapsto \mu \mathsf{X.F}]} \times Z$$

which is derived from the usual *chain rule*: $\partial_A(F \times G) = \partial_A F \times G + F \times \partial_A G$ (see [**AAMG04**]).

For the binary tree data type $\mathsf{Tree}\ A = \mu X.A + A \times X^2$, the partial differentiation technique yields the following calculation (Appendix B.2.1 explains the calculation steps in more detail):

$$
\begin{aligned}
\partial_A(\mathsf{Tree}\ A) \equiv\ & \mu Z.\partial_A(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\ A]} + \partial_X(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\ A]} \times Z\\
\equiv\ & \mu Z.1 + (\partial_A A \times X^2 + A \times \partial_A X^2)_{[X \mapsto \mathsf{Tree}\ A]} + \partial_X(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\ A]} \times Z\\
\equiv\ & \mu Z.1 + (\mathsf{Tree}\ A)^2 + A \times 2 \times \mathsf{Tree}\ A \times Z\\
\equiv\ & (\mu Z.1 + A \times 2 \times \mathsf{Tree}\ A \times Z) \times (1 + (\mathsf{Tree}\ A)^2)\\
\equiv\ & (\mathsf{Path}\ A) \times (1 + (\mathsf{Tree}\ A)^2)
\end{aligned}
$$

A comonadic data type is then provided by $A \times \partial_A(\mathsf{Tree}\ A)$ which is equivalent to the TZ type:

$$
\begin{aligned}
A \times \partial_A(\mathsf{Tree}\ A) \equiv\ & A \times (\mathsf{Path}\ A) \times (1 + (\mathsf{Tree}\ A)^2)\\
\equiv\ & (\mathsf{Path}\ A) \times (A + A \times (\mathsf{Tree}\ A)^2)\\
\equiv\ & (\mathsf{Path}\ A) \times (\mathsf{Tree}\ A) \quad \cong\ \mathit{TZ}\ A
\end{aligned}
$$

Relation to pointed trees. The zipper and pointed tree data types are isomorphic. Proof of this isomorphism follows by showing that the inductive path structure is isomorphic to the inductive definition of a list of booleans, and that the path preserves the rest of the tree structure.

The pointed approach is easier for programming, and provides an efficient *extend*. However, the zipper approach has a more efficient lookup ($\in \mathcal{O}(1)$) compared to the pointer approach ($\in \mathcal{O}(\log n)$). Furthermore, the zipper may be preferred if a program also uses insert, delete, update operations significantly, which for the zipper are also $\in \mathcal{O}(1)$.

3.2.3. Lists

Example 3.1.3 showed the non-empty list comonad, with functor $\mathsf{NEList}A = \mu X.A + A \times X$. There are a number of different possible comonads for non-empty lists with different cursors and accessibility between contexts.

Rotation lists. Similarly to fixed length-lists in Section 3.2.1, the cursor can be taken as the first element in the list where comultiplication rotates the list left through the cursor position, *e.g.* $\delta[1,2,3] = [[1,2,3],[2,3,1],[3,1,2]]$. Example E.2.4 (p. 223) in the appendix gives a definition.

Suffix and prefix lists. Example 3.1.3 showed such a comonad, where only those contexts that are greater than, or equal to, the current context are accessible. This defines the *suffix list comonad* where, for example, $\delta\,[1,2,3] = [[1,2,3],[2,3],[3]]$ with the head of the list as the cursor (see appendix, Example E.2.3, p. 223).

Alternatively, a *prefix list comonad* can be defined by defining the cursor as the last element of the list, *i.e.*, $\varepsilon\ xs = tail\ xs$, and defining accessibility as only those contexts that are less than, or equal to, the current, *e.g.* $\delta\ [1,2,3] = [[1],[1,2],[1,2,3]]$ (see Example E.2.3, p. 223).

Pointed lists. A list can be *pointed* with the type $\mathsf{PList}\,A = [A] \times \mathbb{N}$ which has an *unsafe* cursor (*i.e.* the type allows cursor outside the domain of the list), with definition:

```
data PList a = PList [a] Int   -- pre-condition: [a] non-empty, 0 ⩽ Int < length of the list

instance Comonad PList where
  current (PList xs n)  = xs !! n
  extend k (PList xs n) = PList (map (λn' → k (PList xs n')) [0 .. (length xs − 1)]) n
```

Alternatively, a safe pointed list can be described by a dependent type $\mathsf{PList\,A} = \prod n : \mathbb{N}.A^n \times n$ where lists are finite sequences of A values paired with an index from the finite type of natural numbers $< n$. In Haskell, safe, pointed lists can be defined using type-level natural numbers to index a list data type by its length. This is shown in Appendix E.2.5.

Note that the pointed list allows all contexts to be accessible from any other context, which contrasts with prefix and suffix lists which allowed only a subset of contexts to be accessed.

List zippers. A non-empty list type zipper can be derived using McBride's differentiation technique, $\mathsf{NEList'}A = A \times \partial_A(\mathsf{NEList}A)$ where $\partial_A(\mathsf{NEList}A) = \mathsf{List}\,A \times \mathsf{List}\,A$ (a pair of possibly empty lists). Appendix B.2.2 shows the derivation. The non-empty list-zipper comonad therefore pairs this zipper with a focus value, thus the first list provides a path to the cursor element, and the second provides the elements after the cursor. The zipper is reminiscent of the tree zipper, but with only *left* and *right* navigation operations. The list zipper is equivalent to the pointed version. Appendix E.2.6 gives a Haskell implementation of the list zipper comonad.

3.2.4. Streams and Lucid

Streams (*infinite* lists) have a number of different possible comonad definitions (essentially infinite versions of the above list comonads). A standard definition of streams uses the greatest-fixed point $\mathsf{Str}A = \nu X.A \times X$, (a *codata* type [**Kie99**]), which can be encoded in Haskell as:

```
data Stream a = a :< (Stream a)
```

Uustalu and Vene showed that the Lucid dataflow language can be given a semantics in terms of a stream comonad, and provided an interpreter for higher-order Lucid programs [**UV06**].

Lucid. As described in Chapter 1, Lucid provides a declarative view of imperative, iterative programs [**WA85**]. In an imperative program, "variables" are mutable memory cells rather than variables in the traditional mathematical sense (as in this dissertation). The value of a variable may therefore change throughout a program. Thus, the value of a variable depends on its temporal context (*i.e.*, the *program step*). Lucid declares variables once for a whole program, defining them *intensionally* in terms of all the extensional values of the variable throughout the program. These definitions are typically recursive, encoding iteration by guarded recursion. Lucid programs can therefore be seen as systems of recursive stream equations, where all expressions are streams and variables are defined in terms of their entire history [**WA85, AFJW95**].

Constants in Lucid are constant streams, and many standard arithmetic, logical, and conditional operations are applied *pointwise*, for example, using an informal stream notation:

$$[\![c]\!] = \langle c, c, c, \ldots \rangle$$
$$[\![x + y]\!] = \langle x_0 + y_0, x_1 + y_1, \ldots \rangle$$

Two key *intensional* combinators, that allow context-dependent behaviour, are *next* and *fby* (pronounced *followed by*), which have the following behaviour:

$$[\![next\ x]\!] = \langle x_1, x_2, \ldots \rangle$$
$$[\![x\ fby\ y]\!] = \langle x_0, y_0, y_1, \ldots \rangle$$

Thus, *next* performs a lookahead in a stream and *fby* delays a stream by the first element of another, used for guarded recursion. For example, the stream of natural numbers can be defined recursively in Lucid as: $n = 0\ fby\ (n + 1)$, thus $[\![n]\!] = \langle 0, 1, 2, \ldots \rangle$. The *fby* operation is particularly interesting since its behaviour depends directly on the cursor. This can be seen more clearly in the following definition of *fby* at a particular position/context i:

$$[\![x\ fby\ y]\!]_i = \begin{cases} x_0 & i \equiv 0 \\ y_{i-1} & otherwise \end{cases}$$

Uustalu and Vene defined a number of different stream comonads for the semantics of Lucid-like dataflow languages, categorising them by the *causality* properties allowed for their operations, implied by accessibility between contexts. The *causal* stream comonad only makes available to stream functions values either at or preceding the cursor, the *anti-causal* stream comonad makes available only the current and future values, and the *general* stream comonad allows causality in both directions [**UV05, UV06**].

Anti-causal streams. The anti-causal stream comonad is similar to the suffix-list comonad (Section 3.2.3), modulo the base case which is absent for infinite streams: [**UV06**, p.17]

$$current\ (a :< \_) = a$$
$$extend\ k\ s@(a :< as) = (k\ s) :< (extend\ k\ as)$$

CoKleisli functions of this comonad many depend only on elements at the present position and future positions (hence anti-causal) since *extend* applies a coKleisli function to successive suffix streams. This comonad does not permit *fby* to be defined as a coKleisli function since *fby* depends on its contextual position relative to the start of the stream (which is unknown here) and access to past elements (which are not available here).

Causal streams. Since streams are bounded at their beginning a causal stream, for which computations can only access the current and previous elements, is equivalent to a non-empty list. A causal stream comonad is therefore equivalent to the *prefix*-list comonad (Section 3.2.3). This comonad permits *fby*, but it does not allow *next* to be defined as a coKleisli morphism since it depends on future values which are not available here.

General streams. General stream comonads allow stream operations to depend on values anywhere within a stream, thus the comultiplication operation of a general stream comonad must preserve all elements of the stream data structure, unlike the causal and anti-causal streams. Thus, a pointed stream (pairing the *Stream* data type with a natural number for the cursor) or stream zipper comonad can be used [**UV06**, p. 18]:

$$\textbf{data}\ \mathit{StreamZ}\ a = \mathit{StreamZ}\ [a]\ a\ (\mathit{Stream}\ a)$$

The definition of the stream zipper comonad resembles that for the list zipper comonad shown in Appendix E.2 (p. 222). In Haskell, the zipper implementation may be preferable to the pointer approach, since lookup is more efficient for the zipper (see the discussion for trees).

Lucid is a higher-order language, with function abstraction and first-class functions, thus further structure than a general stream comonad is required for its semantics, discussed in Section 3.3.3 (p. 70). Chapter 4 shows first-order Lucid programs embedded in Haskell.

As is well known, the functor $\mathsf{F}A = \nu X.A \times X$ is isomorphic to the functor $\mathsf{F}'A = A^{\mathbb{N}}$ (the terminal sequence of $A \times -$). The next two comonads can encode streams using exponents.

3.2.5. Exponents

The *exponent comonad*, with functor $\mathsf{D}A = X \Rightarrow A$ where X is a monoid $(X, \oplus, 0)$,[3] is a comonad where the unit $\oplus$ of the monoid is the current context position and the binary operation $\oplus$ defines the accessibility between contexts. In Haskell, it is defined:

$$\begin{aligned}&\textbf{instance}\ \mathit{Monoid}\ x \Rightarrow \mathit{Comonad}\ ((\rightarrow)\ x)\ \textbf{where}\\ &\quad \mathit{current}\ f = f\ 0\\ &\quad \mathit{extend}\ k\ f = \lambda x \rightarrow k\ (\lambda x' \rightarrow f\ (x\ `\oplus`\ x'))\end{aligned}$$

[3] In Haskell the *Data.Monoid* package provides a monoid class with the binary operation $\oplus$ = *mappend* ::*Monoid* $x \Rightarrow x \rightarrow x \rightarrow x$ and unit element $0 \cong$ *mempty* ::*Monoid* $x \Rightarrow x$

Anti-causal streams. The exponent comonad with monoid $(\mathbb{Z}, +, 0)$ defines a stream comonad, but where the contextual position is unknown. The exponent comonad with monoid $(\mathbb{N}, +, 0)$ is isomorphic to the anti-causal stream comonad from before. In Haskell there is no numerical data type for natural numbers (only integers). An inductive data type representation of the natural numbers could be used instead. Whilst the exponent comonad is general and simple, the *Stream* data type above provides a more efficient implementation since functions are not memoized in Haskell, causing recomputation.

Other finite comonads can be encoded using the exponent comonad, with runtime checks on the contexts, *e.g.*, for a list of length n, its equivalent exponent version is $(\lambda i \to \textbf{if}\ (i < n)\ \textbf{then}\ \ldots\ \textbf{else}\ \mathit{error}\ \texttt{"Out of bounds"})$, where the error is equivalent to an exception from taking the head of an empty-list.

Numerical functions. For the monoid $(\mathbb{R}, +, 0)$, the exponent comonad can be understood as capturing *curves* (*numerical functions*). An example operation is *numerical differentiation*:

$$\mathit{differentiate} :: (\mathit{Double} \to \mathit{Double}) \to \mathit{Double}$$
$$\mathit{differentiate}\ f = (f\ 0.01 - f\ 0.0)\ /\ 0.01$$

Since *extend* passes f shifted by x' to *differentiate* then $f\ 0.0$ computes $f\ x'$. The value by which f is shifted is unknown within a local operation.

Laziness comonad. In an eagerly evaluated language, such as ML, lazy evaluation, *i.e.*, delaying computations until demanded, can be programmed manually by wrapping expressions in a λ-term matching a *unit* value (written () here), *i.e.*, $(\lambda().e : () \to \tau)$. However, programming with manual laziness constructions is verbose and inelegant. For example, consider the following term in an eager language, where a delayed function $f : () \to (\sigma \to \tau)$ is lazily applied to a delayed argument $e : () \to \sigma$:

$$\lambda().(f())(e()) : () \to \tau$$

Previously, Brookes and Geva mentioned that comonads provide a mechanism for describing demand-driven (lazy) computations [**BG91**]. Indeed, the boilerplate code for lifting expressions to lazy computations and composing lazy expressions is captured by a *computational comonad* with the functor $\mathsf{Lazy}A = () \Rightarrow A$. Computational comonads provide an operation $\eta : A \to \mathsf{D}A$ [**BG91**]. The comonad operations are then:

$\eta\ e = \lambda().e$ (lift eager to lazy computations)
$\varepsilon\ l = l()$ (evaluate a computation)
$f^\dagger\ l = \lambda().f l$ (lazily compute from a lazy computation)

Relationship to *intensions*. The introductory chapter briefly described the field of *intensional logic*, prompted by the role of context in languages. Carnap introduced the term *intension* to describe the entire meaning of a term, encapsulating its meaning in all *contexts*. He formulated this mathematically as a function from states (or contexts) to *extensional* values [**Car47**]. The

exponent data type here therefore captures Carnap's model of intensions (or context-dependent values) where the comonad describes how to compose intensional fragments.

3.2.6. *In context* comonad

Whilst the exponent comonad abstracts various comonads, the following abstracts various comonads whose type is paired with a cursor therefore removing the need for the monoid structure. This comonad is well known, and is called the "in context" comonad here, defined:

$$\textbf{data}\ \mathit{InContext}\ c\ a = \mathit{InContext}\ (c \to a)\ c$$

$$\textbf{instance}\ \mathit{Comonad}\ (\mathit{InContext}\ c)\ \textbf{where}$$
$$\quad \mathit{current}\ (\mathit{InContext}\ f\ c) = f\ c$$
$$\quad \mathit{extend}\ k\ (\mathit{InContext}\ f\ c) = \mathit{InContext}\ (\lambda c' \to k\ (\mathit{InContext}\ f\ c'))\ c$$

Thus the context paired with the function is the cursor (as defined in *current*), and *extend* returns a contextual computation which at a c' applies the local operation to parameter computation at c'. Thus the result of comultiplication has inner computations which are *focussed* (*i.e.*, have their cursor set) at the context described by the outer contextual computation.

This comonad is sometimes called the *costate* comonad due to its relation to the state monad, which is induced from the same pair of adjoint functors. The product functor $DX = A \times X$, which is left adjoint to the exponent functor $TX = X \Rightarrow A$, thus $D \dashv T$, induces the *state* monad $TDA = X \Rightarrow (A \times X)$ and the *costate* comonad $DTA = (X \Rightarrow A) \times X$.

General streams. A general stream comonad with a manifest cursor is provided by instantiating the *InContext* comonad with natural number contexts, *i.e.*, the functor $\mathsf{Stream}\ A = A^{\mathbb{N}} \times \mathbb{N}$. Note that, local operations for this comonad are isomorphic (by uncurrying) to functions of type $A^{\mathbb{N}} \to A^{\mathbb{N}}$; thus the local operations capture general stream functions.

Similarly to the exponent comonad, finite comonads with a cursor may be encoded using a domain membership test, *e.g.*, $\mathit{InContext}\ (\mathit{Int}, \mathit{Int})$ captures two-dimensional array comonads, where a finite array might include a runtime bounds-check, *e.g.*, for bounds bx and by:

$$x = (\mathit{InContext}\ (\lambda(x, y) \to \textbf{if}\ (x \leqslant bx \wedge y \leqslant by)\ \textbf{then}\ ...\ \textbf{else}\ \mathit{error}\ \texttt{"Out of bounds"})\ (0, 0)$$

3.2.7. Composing comonads

Two comonads can be composed if there exists a *distributive law* between the two.

Definition 3.2.1. A *distributive law* between two comonads $(\mathsf{D}, \varepsilon, \delta)$ and $(\mathsf{E}, \varepsilon', \delta')$ is a natural transformation $\sigma : \mathsf{ED} \dot{\to} \mathsf{DE}$ satisfying the following axioms of preservation of the comonad operations by σ:

$$[\sigma 1]\quad \varepsilon\mathsf{E} \circ \sigma = \mathsf{E}\varepsilon \qquad [\sigma 3]\quad \sigma\mathsf{E} \circ \mathsf{E}\sigma \circ \delta' = \mathsf{D}\delta' \circ \sigma$$
$$[\sigma 2]\quad \mathsf{D}\varepsilon' \circ \sigma = \varepsilon'\mathsf{D} \qquad [\sigma 4]\quad \mathsf{D}\sigma \circ \sigma\mathsf{D} \circ \mathsf{E}\delta = \delta\mathsf{E} \circ \sigma$$

Lemma 3.2.2. *Given two comonads* $(\mathsf{D}, \varepsilon, \delta)$ *and* $(\mathsf{E}, \varepsilon', \delta')$ *and a distributive law* $\lambda : \mathsf{ED} \dot{\to} \mathsf{DE}$ *then there is a comonad* $(\mathsf{ED}, \varepsilon'', \delta'')$ *with:*

– $\varepsilon'' := \ \mathsf{ED} \xrightarrow{\varepsilon'\mathsf{D}} \mathsf{D} \xrightarrow{\varepsilon} 1$ *(*$\equiv \varepsilon' \circ \mathsf{E}\varepsilon$*, by naturality).*

– $\delta'' := \ \mathsf{ED} \xrightarrow{\delta'\mathsf{D}} \mathsf{EED} \xrightarrow{\mathsf{EE}\delta} \mathsf{EEDD} \xrightarrow{\mathsf{E}\sigma\mathsf{D}} \mathsf{EDED}$ *(*$\equiv \mathsf{E}\sigma\mathsf{D} \circ \delta'\mathsf{DD} \circ \mathsf{E}\delta$*, by naturality).*

The comonad axioms for the composite follow from the axioms of the distributive law and the axioms of the underlying comonads.

Example 3.2.3. A comonad is *strong* if there is a tensorial strength $\mathsf{st}_{A,X} : (\mathsf{D}A) \times X \to \mathsf{D}(A \times X)$ that commutes with the comonad operations (dual to strong monads, see Appendix B.1.4 (p. 188)). This tensorial strength corresponds to a distributive law between D and the product comonad where the strong comonad axioms correspond to axioms [σ1],[σ4] of the distributive law and [σ2],[σ3] follow from the strong functor axioms.

Appendix E.2.7 shows an encoding for composing comonads in Haskell.

3.2.8. Cofree comonad

The *cofree comonad* captures a comonad for a general notion of a branching data structure with $\mathsf{CoFree}A = \nu X.A \times \mathsf{F}X$, where F provides a branching structure and each "fork" in the branch is labelled [**UV08**, **UV06**]. It has a simple definition where the root of the tree is the current context. In Haskell it can be defined as follows:

```
data Cofree f a = Root {label :: a, branch :: f (Cofree f a)}

instance Comonad (Cofree f) where
  current (Root a _) = a
  extend f (Root a ts) = Root (f (Root a ts)) (fmap (extend f) ts)
```

Examples of F include:

– $\mathsf{F} = 1$ (identity functor $\mathsf{F}U = U$) – anti-causal streams;
– $\mathsf{F} = \Delta$ (diagonal functor $\mathsf{F}U = U \times U$) – infinite binary suffix trees;
– $\mathsf{F}U = 1 + U \times U$ – possibly finite binary suffix trees.

Uustalu and Vene have also described the *cofree recursive comonad* on $\mu X.A \times \mathsf{F}X$, although this is not discussed further here (see [**UV02**, **UV11**]). Section 7.4.1 (p. 160) shows a construction of comonads from a particular kind of *coalgebra*, for which the cofree comonad is equivalent to the final coalgebra.

3.3. Categorical semantics of the contextual λ-calculus

The understanding of Lucid as an *intensional language*, where the denotation of an expression depends on some implicit, discrete temporal context (*e.g.*, [**Wad95a**, **AFJW95**, **PP95**]) is made precise by the categorical semantics, given by Uustalu and Vene, in terms of stream comonads [**UV06**]. Uustalu and Vene generalised their approach to other notions of context-dependence, defining a categorical semantics for a context-dependent simply-typed λ-calculus parameterised by a *symmetric (semi)-monoidal comonad* capturing some notion of context [**UV08**].

$$[\![\text{ABS}]\!]\frac{[\![\Gamma, x : \sigma \vdash e : \tau]\!] : \mathsf{D}(\Gamma \times \sigma) \to \tau}{[\![\Gamma \vdash \lambda x.e : \sigma \Rightarrow \tau]\!] : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau)}$$

$$[\![\text{APP}]\!]\frac{[\![\Gamma \vdash e_1 : \sigma \to \tau]\!] : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau) \qquad [\![\Gamma \vdash e_2 : \tau]\!] : \mathsf{D}\Gamma \to \sigma}{[\![\Gamma \vdash e_1\, e_2 : \tau]\!] : \mathsf{D}\Gamma \to \tau}$$

$$[\![\text{VAR}]\!]\frac{x_i : \tau_i \in \Gamma}{[\![\Gamma \vdash x_i : \tau]\!] : \mathsf{D}\Gamma \to \tau_i} \text{ where } (|\Gamma| = n) \wedge (1 \leqslant i \leqslant n)$$

$$[\![\text{LET}]\!]\frac{[\![\Gamma \vdash e_1 : \sigma]\!] : \mathsf{D}\Gamma \to \sigma \qquad [\![\Gamma, v : \sigma \vdash e_2 : \tau]\!] : \mathsf{D}(\Gamma \times \sigma) \to \tau}{[\![\Gamma \vdash \textbf{let } v = e_1 \textbf{ in } e_2]\!] : \mathsf{D}\Gamma \to \tau}$$

Figure 3.5. Outline of the structure of a categorical semantics for a contextual λ-calculus.

Their approach is reworked here following the categorical semantics approach of Section 2.2. A contextual semantics is given to the simply-typed λ-calculus, with *let*-binding, which is referred to as the *contextual λ-calculus* here. The semantics developed here provides a generalisation of the Uustalu-Vene semantics. The necessary additional structure required of a category in order to give a semantics for the simply-typed λ-calculus *modulo $\beta\eta$-equality* is given first in Section 3.3.1 before a discussion of the axioms required on the additional structure for various syntactic properties in Section 3.3.2, which was discussed less extensively by Uustalu and Vene. Thus, this section contributes a systematic construction and analysis of the syntactic properties that follow from different axiomatisations of monoidal comonads.

3.3.1. Categorical semantics modulo $\beta\eta$-equality

The categorical semantics uses a coKleisli category for a comonad D, where types map to objects $[\![\tau]\!] \in \mathbb{C}_{\mathsf{D}0}$ and typing judgments to morphisms:

$$[\![\overline{v : \tau} \vdash e : \tau]\!] : [\![\overline{\tau}]\!] \to [\![\tau]\!] \in \mathbb{C}_{\mathsf{D}1}$$
$$: \mathsf{D}\,[\![\overline{\tau}]\!] \to [\![\tau]\!] \in \mathbb{C}_1$$

In order to provide an intuition for the behaviour of a contextual language, **Figure** 3.5 shows the structure of the semantics by showing the signatures of the morphisms resulting from the interpretation $[\![-]\!]$. Reading the rules bottom-up (from conclusion to premise) indicates the propagation of the comonadic context (encoded by D) of a term to its subterms. In the $[\![\text{ABS}]\!]$ rule, the incoming contextual computation for a function body is obtained by combining in some way the context of the function and the context of the argument. In the $[\![\text{APP}]\!]$ rule, the context is propagated to both the function subterm and the argument subterm. In the $[\![\text{LET}]\!]$ rule, the context is propagated to both the bound sub-term and the receiving sub-term.

Following from the presentation of categorical semantics for the pure simply-typed λ-calculus in Section 2.2, $\mathbb{C}_{\mathsf{D}}$ requires a binary object mapping $\times^{\mathsf{D}}$ (not necessarily a categorical product) for modelling concatenation of free-variable assumptions, with notions of pairing and projections,

a binary object mapping $\Rightarrow^{\mathsf{D}}$ for modelling functions, and currying and uncurrying operations to support abstraction and application (conditions $(A.i)$-$(A.v)$ from Section 2.2.7 (p. 33)), without any equational theory. The requirements of these components elicits further additional structure on the comonad D which is explained below.

As first shown by Bierman and de Paiva in the setting of categorical modal logic, a context which is structured by a comonad (in their case a modal context), *i.e.*, $\mathsf{D}[\![\Gamma]\!]$ where $[\![\Gamma]\!] = A_1 \times \ldots \times A_n$, requires an operation $\mathsf{D}(A_1 \times \ldots \times A_n) \to \mathsf{D}A_1 \times \ldots \times \mathsf{D}A_n$ such that free-variables can be abstracted (or, that hypotheses can be turned into antecedents of implications) [**BdP96**]. This operation is provided by a *(lax) monoidal functor*.

Definition 3.3.1. A functor $\mathsf{F} : \mathbb{C} \to \mathbb{D}$, between two monoidal categories $(\mathbb{C}, \otimes, I_{\mathbb{C}})$ and $(\mathbb{D}, \bullet, I_{\mathbb{D}})$, is *lax monoidal* if it has the following natural transformation and morphism:

$$\mathsf{m}_{A,B} : \mathsf{F}A \bullet \mathsf{F}B \to \mathsf{F}(A \otimes B)$$
$$\mathsf{m}_1 : I_{\mathbb{D}} \to \mathsf{F}I_{\mathbb{C}}$$

satisfying a number of coherence conditions such that $\mathsf{m}_{A,B}$ and m_1 preserve the associativity and unit natural transformations for $\mathbb{C}$ and $\mathbb{D}$ (shown in Appendix B.1.2 (p. 185)) and thus $\mathsf{m}_{A,B}$ and m_1 act as monoid where m_1 creates the *unit*. Another view is that $\mathsf{m}_{A,B}$ and m_1 witness that F is a (lax) homomorphism, preserving the monoidal structure between $\mathbb{C}$ and $\mathbb{D}$.

There are number of variations of monoidal functors used later:

– A *colax monoidal functor* (sometimes called *lax comonoidal* or *oplax monoidal*) has dual operations to a lax monoidal functor:

$$\mathsf{n}_{A,B} : \mathsf{F}(A \otimes B) \to \mathsf{F}A \bullet \mathsf{F}B$$
$$\mathsf{n}_1 : \mathsf{F}I_{\mathbb{C}} \to I_{\mathbb{D}}$$

  Any functor on a category with finite products is canonically colax monoidal with respect to the monoidal structure of products. This is shown later in Lemma 3.3.5 (p. 66).
– A *strong monoidal functor* is a monoidal functor where $\mathsf{m}_{A,B}$ and m_1 are isomorphisms, thus a strong monoidal functor is both lax and colax, where $\mathsf{m}_{A,B}^{-1} = \mathsf{n}_{A,B}$ and $\mathsf{m}_1^{-1} = \mathsf{n}_1$.
– A *semi-monoidal functor* does not have the m_1 morphism.
– A *symmetric monoidal functor* maps between symmetric monoidal categories, preserving the symmetry transformations of the source and target categories.

Example 3.3.2. The List endofunctor $\mathsf{List}A = \nu X.1 + A \times X$, using the greatest fixed-point, is a monoidal functor where $\mathsf{m}_{A,B}$ is the *zip* operation for lists, *e.g.*, $\mathsf{m}_{A,B}[x_0, x_1, x_2]\ [y_0, y_1, y_2, y_3] = [(x_0, y_0), (x_1, y_1), (x_2, y_2)]$, taking the intersection of the *shapes* of the parameter list, and m_1 is the infinite list $\mathsf{m}_1\,() = [(), (), (), \ldots]$ (for the terminal object $()$).

Example 3.3.3. The List endofunctor is colax monoidal where $\mathsf{n}_{A,B}$ is the *unzip* operation for lists, *e.g.*, $\mathsf{n}_{A,B}\,[(x_0, y_0), (x_1, y_1), (x_2, y_2)] = [[x_0, x_1, x_2], [y_0, y_1, y_2]]$ and n_1 trivially returns the terminal object $()$.

In the following semantics, monoidal functors, and some of the above variations, are required to provide the necessary structures for the semantics and equational theory. The underlying category $\mathbb{C}$ is assumed to be a CCC with products $\times$ and exponents $\Rightarrow$. The minimal requirements for a coKleisli category $\mathbb{C}_{\mathsf{D}}$ such that it provides a categorical semantics for the simply-typed λ-calculus modulo any equational theory, are described below.

Free-variable contexts. (Section 2.2.7, *A.ii*) For every $A, B \in \mathbb{C}_{\mathsf{D}0}$ the object $(A \times^{\mathsf{D}} B) \in \mathbb{C}_{\mathsf{D}0}$ is defined as the object $A \times B \in \mathbb{C}_0$, *i.e.*, using the products of the underlying category. Pairing is defined by pairing in the underlying category, where for all $f : \mathsf{D}Z \to X, g : \mathsf{D}Z \to Y \in \mathbb{C}_1$:

$$\langle f, g \rangle^{\mathsf{D}} = \langle f, g \rangle : \mathsf{D}Z \to X \times Y \in \mathbb{C}_1$$

with projections $\pi_1^{\mathsf{D}} = \pi_1 \circ \varepsilon : \mathsf{D}(A \times B) \to A$ and $\pi_2^{\mathsf{D}} = \pi_2 \circ \varepsilon : \mathsf{D}(A \times B) \to B$. This construction implies the usually properties of categorical products (Definition 2.2.3 (p. 28)).

Functions. (Sect. 2.2.7, *A.iii*) For all $A, B \in \mathbb{C}_{\mathsf{D}0}$ then $(A \Rightarrow^{\mathsf{D}} B) \in \mathbb{C}_{\mathsf{D}0}$ is defined as the object $(\mathsf{D}A \Rightarrow B) \in \mathbb{C}_0$.

Abstraction. (currying) (Sect. 2.2.7, *A.iv*) The currying operation for $\mathbb{C}_{\mathsf{D}}$, between $\times^{\mathsf{D}}$ and $\Rightarrow^{\mathsf{D}}$ can then be defined if D is a lax (semi-)monoidal functor over $\times$, where:

$$\Phi_{A,B} = \mathbb{C}(\mathsf{D}(A \times X), B) \xrightarrow{\mathbb{C}(\mathsf{m}_{A,X}, -)} \mathbb{C}(\mathsf{D}A \times \mathsf{D}X, B) \xrightarrow{\phi_{A,B}} \mathbb{C}(\mathsf{D}A, \mathsf{D}X \Rightarrow B)$$

i.e., in pointed-style, $\Phi_{A,B}(f : \mathsf{D}(A \times X) \to B) = \phi_{A,B}(f \circ \mathsf{m}_{A,X}) : \mathsf{D}A \to (\mathsf{D}X \Rightarrow B)$.

An intuition for the use of $\mathsf{m}_{A,B}$ here is that it combines, or merges, two contexts, which in the semantics combines the context of the declaration site of a function with the context of the application site for the function to give the context of the function body.

Application. (uncurrying) (Sect. 2.2.7, *A.v*) Uncurrying can be defined if D is *colax monoidal* with $\mathsf{n}_{A,B} : \mathsf{D}(A \times B) \to \mathsf{D}A \times \mathsf{D}B$ as follows:

$$\Psi_{A,B} = \mathbb{C}(\mathsf{D}A, \mathsf{D}X \Rightarrow B) \xrightarrow{\phi^{-1}_{A,B}} \mathbb{C}(\mathsf{D}A \times \mathsf{D}X, B) \xrightarrow{\mathbb{C}(\mathsf{n}_{A,X}, -)} \mathbb{C}(\mathsf{D}(A \times X), B)$$

i.e., in pointed-style, $\Psi_{A,B}(g : \mathsf{D}A \to (\mathsf{D}X \Rightarrow B)) = \phi^{-1}_{A,B}(g) \circ \mathsf{n}_{A,X} : \mathsf{D}(A \times X) \to B$.

An intuition for the colax monoidal operation $\mathsf{n}_{A,B}$ here is that it splits a context into two, which in the semantics for application provides the context for the function and the context for the argument of the function.

Figure 3.6 shows the usual semantics for the simply-typed λ-calculus, specialised to the additional structures defined above for $\mathbb{C}_{\mathsf{D}}$.

Alternative definition. Uustalu and Vene's approach differs in their semantics of application:

$$[\![\text{APP}]\!] \frac{[\![\Gamma \vdash e_1 : \sigma \to \tau]\!] = k_1 : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau) \qquad [\![\Gamma \vdash e_2 : \sigma]\!] = k_2 : \mathsf{D}\Gamma \to \sigma}{[\![\Gamma \vdash e_1\, e_2 : \tau]\!] = \mathsf{ev} \circ \langle k_1, k_2^{\dagger} \rangle : \mathsf{D}\Gamma \to \tau} \tag{30}$$

where ev is the usual morphism provided by the categorical exponents of the base category $\mathbb{C}$. This definition is equivalent to the semantics in **Figure** 3.6 (which used $\mathsf{n}_{A,B}$) if $\mathsf{n}_{A,B}$ is the operation of a colax *idempotent* monoidal functor.

Definition 3.3.4. A (co)lax monoidal functor is *idempotent* if the following commutes:

$$\begin{array}{ccc} \mathsf{F}A & \xrightarrow{\Delta} & \mathsf{F}A \times \mathsf{F}A \\ & \searrow^{\mathsf{F}\Delta} & \uparrow \mathsf{n}_{A,A} \quad \downarrow \mathsf{m}_{A,A} \\ & & \mathsf{F}(A \times A) \end{array} \tag{31}$$

following the $\mathsf{n}_{A,A}$ arrow for the colax case and $\mathsf{m}_{A,A}$ arrow for the lax case.

The equivalence between $[\![\text{APP}]\!]$ in **Figure** 3.6 and the refined semantics in equation (30) is then:

$$\begin{array}{lll} & \phi^{-1}k_1 \circ \mathsf{n}_{\Gamma,\sigma} \circ \langle \varepsilon, k_2 \rangle^\dagger & \\ = & \phi^{-1}k_1 \circ \mathsf{n}_{\Gamma,\sigma} \circ \mathsf{D}(\varepsilon \times k_2) \circ \mathsf{D}\Delta \circ \delta & \{\mathsf{D}f \circ \delta = f^\dagger \textit{ and } \times \textit{ universal property}\} \\ = & \phi^{-1}k_1 \circ (\mathsf{D}\varepsilon \times \mathsf{D}k_2) \circ \mathsf{n}_{\mathsf{D}\Gamma,\mathsf{D}\Gamma} \circ \mathsf{D}\Delta \circ \delta & \{\mathsf{n}_{A,B} \textit{ naturality}\} \\ = & \phi^{-1}k_1 \circ (\mathsf{D}\varepsilon \times \mathsf{D}k_2) \circ \Delta \circ \delta & \{\mathsf{n}_{A,B} \textit{ idempotence } (31)\} \\ = & \phi^{-1}k_1 \circ ((\mathsf{D}\varepsilon \circ \delta) \times (\mathsf{D}k_2 \circ \delta)) \circ \Delta & \{\Delta \textit{ naturality }\} \\ = & \phi^{-1}k_1 \circ (id \times (\mathsf{D}k_2 \circ \delta)) \circ \Delta & [\text{C2}] \textit{ (comonad law)} \\ = & \mathsf{ev} \circ (k_1 \times id) \circ (id \times (\mathsf{D}k_2 \circ \delta)) \circ \Delta & \{\phi^{-1}f = \mathsf{ev} \circ (f \times id) \textit{ adjunction definitions}\} \\ = & \mathsf{ev} \circ (k_1 \times (\mathsf{D}k_2 \circ \delta)) \circ \Delta & \{\times \textit{ functoriality}\} \\ = & \mathsf{ev} \circ \langle k_1, k_2^\dagger \rangle & \{\mathsf{D}f \circ \delta = f^\dagger \textit{ and } \times \textit{ universal property}\} \end{array}$$

For any (semi-)monoidal comonad over a category with finite categorical products, there is a canonical idempotent colax monoidal structure:

$$[\![\text{ABS}]\!]\frac{[\![\Gamma, x : \sigma \vdash e : \tau]\!] = k : \mathsf{D}(\Gamma \times \sigma) \to \tau}{\begin{array}{r}[\![\Gamma \vdash \lambda x.e : \sigma \Rightarrow \tau]\!] = \Phi\, k : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau) \\ = \phi(k \circ \mathsf{m}_{\Gamma,\sigma}) : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau)\end{array}}$$

$$[\![\text{APP}]\!]\frac{[\![\Gamma \vdash e_1 : \sigma \to \tau]\!] = k_1 : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau) \qquad [\![\Gamma \vdash e_2 : \tau]\!] = k_2 : \mathsf{D}\Gamma \to \sigma}{\begin{array}{r}[\![\Gamma \vdash e_1\, e_2 : \tau]\!] = \Psi k_1 \circ^{\mathsf{D}} \langle id^{\mathsf{D}}, k_2 \rangle^{\mathsf{D}} : \mathsf{D}\Gamma \to \tau \\ = \phi^{-1}k_1 \circ \mathsf{n}_{\Gamma,\sigma} \circ \langle \varepsilon, k_2 \rangle^\dagger : \mathsf{D}\Gamma \to \tau\end{array}}$$

$$[\![\text{VAR}]\!]\frac{x_i : \tau_i \in \Gamma}{\begin{array}{r}[\![\Gamma \vdash x_i : \tau]\!] = \pi_i^{n\mathsf{D}} : \mathsf{D}\Gamma \to \tau_i \\ = \pi_i^n \circ \varepsilon : \mathsf{D}\Gamma \to \tau_i\end{array}} \text{ where } (|\Gamma| = n) \wedge (1 \leqslant i \leqslant n)$$

$$[\![\text{LET}]\!]\frac{[\![\Gamma \vdash e_1 : \sigma]\!] = k_1 : \mathsf{D}\Gamma \to \sigma \qquad [\![\Gamma, v : \sigma \vdash e_2 : \tau]\!] = k_2 : \mathsf{D}(\Gamma \times \sigma) \to \tau}{\begin{array}{r}[\![\Gamma \vdash \textbf{let } v = e_1 \textbf{ in } e_2]\!] = k_2 \circ^{\mathsf{D}} \langle id^{\mathsf{D}}, k_1 \rangle^{\mathsf{D}} : \mathsf{D}\Gamma \to \tau \\ = k_2 \circ \langle \varepsilon, k_1 \rangle^\dagger : \mathsf{D}\Gamma \to \tau\end{array}}$$

Figure 3.6. Categorical semantics for a contextual simply-typed λ-calculus.

Lemma 3.3.5. *Every endofunctor* F *over a category with finite (categorical) products has a unique (canonical) colax monoidal structure with the operations* $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$ *and* $\mathsf{n}_1 = !_{\mathsf{F}1}$.

Proof. (sketch – full proof in Appendix C.2.1, p. 196) Coherence conditions of a colax monoidal functor (Definition B.1.5, p. 187) for $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$ and $\mathsf{n}_1 = !_{\mathsf{F}1}$ follow from the universal property of products and the monoidal natural transformations of $\mathbb{C}$.

Assume the existence of some other colax monoidal structure $\mathsf{n}'_{A,B}$ and n'_1. The unital properties of n'_1 with respect to $\mathsf{n}'_{A,B}$ and the canonical definitions of the unital natural transformations $\lambda : 1 \times A \to A = \pi_2$ and $\rho : A \times 1 \to A = \pi_1$ (Example B.1.3, p. 186) imply the properties:

$$\mathsf{F}\pi_1 = \pi_1 \circ \mathsf{n}'_{X,Y} \qquad \mathsf{F}\pi_2 = \pi_2 \circ \mathsf{n}'_{X,Y} \tag{32}$$

from which $\mathsf{n}'_{X,Y} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$. Thus, the canonical $\mathsf{n}_{A,B}$ is unique by contradiction. The unit operation $\mathsf{n}_1 = !_{\mathsf{F}1}$ is trivially unique by the universal property of the terminal morphism.

□

While the signature $\mathsf{n}_{A,B} : \mathsf{D}(A \times B) \to \mathsf{D}A \times \mathsf{D}B$ suggests that $\mathsf{n}_{A,B}$ may divide/split the context encoded by its parameter, the canonical definition of $\mathsf{n}_{A,B}$ simply *duplicates* the context encoded by its parameter.

Proposition 3.3.6. *For an endofunctor* F *over a category with finite products, the following two properties are equivalent (proof given in Appendix C.2.1, p. 197):*

(i) $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$
(ii) $\mathsf{n}_{A,B} \circ \mathsf{F}\Delta = \Delta$ *(i.e.,* F *is idempotent colax semi-monoidal)*

Thus, the canonical colax monoidal functor is also idempotent. Uustalu and Vene's semantics is therefore equivalent to the refined semantics shown here when $\mathsf{n}_{A,B}$ is the canonical definition.

Whilst a colax monoidal functor (in a category with products) must be the unique, canonical definition, there are many possible colax *semi*-monoidal functors (without unit operation n_1) since the proof of uniqueness for the canonical definition relies on colax unitality properties. In the refined semantics here there is no requirement for a full colax monoidal functor with both $\mathsf{n}_{A,B}$ and n_1, only a colax *semi*-monoidal functor with $\mathsf{n}_{A,B}$. Therefore, the refined semantics is more general, where $\mathsf{n}_{A,B}$ can produce two different contexts for the semantics of application, rather than duplicating the incoming context. The following example uses this generality.

Example 3.3.7. Consider an implementation of a call-by-name (CBN) λ-calculus using a *call stack*, which may have a practical size limit (after which a stack overflow occurs). The call stack forms part of the context of an execution, where β-reduction grows the stack.

Consider a term stack in this calculus which returns the current stack depth as an integer. The stack expression is *impure*, violating $\beta\eta$-equality, since the following terms produce different results:, $\mathsf{stack} \rightsquigarrow 0$, $(\lambda x.\mathsf{stack})() \rightsquigarrow 1$, and $\lambda y.((\lambda x.\mathsf{stack})())y \rightsquigarrow 2$.

The semantics of this calculus can be modelled by the product comonad $\mathsf{D}A = A \times \mathbb{Z}$ (where the second component records the stack depth) and a lax and colax semi monoidal functor with:

$$\mathsf{m}_{A,B}\left((x,d),(y,c)\right) = ((x,y),c+d)$$
$$\mathsf{n}_{A,B}\left((x,y),r\right) = ((x,1),(y,r))$$

(note that $\mathsf{n}_{A,B}$ is not the canonical definition of $\mathsf{n}_{A,B}$, neither is it idempotent). Therefore, $\mathsf{m}_{A,B}$ (for abstraction) computes the stack depth of a function body as the stack depth of its call site c plus the stack depth of its definition site d; $\mathsf{n}_{A,B}$ (for application) takes the current stack depth as the stack depth in the right context (for the argument) and just a depth of one for left (the function). This corresponds to the following reduction rule in CBN:

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2}$$

where the left-hand side is reduced first (*i.e.*, at a depth of 1). Once the left-hand side is reduced to a λ-abstraction, a β-reduction occurs which results in the body of the function having stack depth $r+1$ (from the semantics of $\mathsf{n}_{A,B}$ followed by $\mathsf{m}_{A,B}$). For example,

$$((\lambda x.x)(\lambda f.\mathsf{stack}))() \rightsquigarrow 1$$

since the left-hand side of the application is evaluated first to $(\lambda f.\mathsf{stack})$ before β-reduction of the outer application occurs, the use of stack evaluates to 1. As another example,

$$(\lambda f.(\lambda x.f\ ())\ ())\ (\lambda x.\mathsf{stack}) \rightsquigarrow 3$$

A limited stack depth can then be modelled by placing a bound in application:

$$\mathsf{n}_{A,B}\left((x,y),r\right) = \begin{cases} \textit{throw stack overflow exception} & (r+1) \geqslant \textit{limit} \\ ((x,1),(y,r)) & \textit{otherwise} \end{cases}$$

This example is not possible with the canonical definition of $\mathsf{n}_{A,B}$ which would duplicate the stack depth leading to a different semantics.

Summary. Modulo any equational theory, a semantics for the simply-typed λ-calculus on top of a coKleisli category $\mathbb{C}_{\mathsf{D}}$ (summarised in **Figure** 3.6) requires the additional structures of a lax and colax semi-monoidal functor with operations:

- $\mathsf{m}_{A,B} : \mathsf{D}A \times \mathsf{D}B \to \mathsf{D}(A \times B)$, used for λ-abstraction which combines two contexts coming from the defining and application sites,
- $\mathsf{n}_{A,B} : \mathsf{D}(A \times B) \to \mathsf{D}A \times \mathsf{D}B$, used for application, which splits the incoming context at the application site into two separate contexts.

Lemma 3.3.5 and its proof show that the canonical definition $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$, $\mathsf{n}_1 = !_{\mathsf{F}1}$ is the unique *colax monoidal* functor, but not necessarily the unique colax *semi*-monoidal structure. The uniqueness proof relied on the coherence conditions of the unit operation n_1 (74) (p. 187) — if no such conditions hold, *i.e.*, if F is colax *semi*-monoidal without units, then there may be other valid colax semi-monoidal operations $\mathsf{n}_{A,B}$. The semantics here is therefore more general

that Uustalu and Vene's, where any colax monoidal definition may be used. This generality was demonstrated in Example 3.3.7. The semantics here collapses to the Uustalu-Vene semantics if the colax monoidal operation has the canonical definition $\mathsf{n}_{A,B} = \langle \mathsf{D}\pi_1, \mathsf{D}\pi_2 \rangle$. The next section considers additional properties of lax and colax monoidal functors which model $\beta\eta$-equational theory for the contextual λ-calculus.

Note that, an implementation of a contextual language requires a runtime system which provides an encoding of the top-level context. That is, for a closed term $[\![\emptyset \vdash e : \tau]\!] : \mathsf{D}() \to [\![\tau]\!]$, a runtime system must provide an intension $\mathsf{D}()$ for the incoming context. In Lucid, open terms are allowed as programs, where free-variables are inputs and the user is prompted on `stdin` for elements of the input stream of variables at particular context as they are lazily demanded.

3.3.2. Additional structure required for $\beta\eta$-equality

Section 2.2.7 summarised the additional conditions required of a categorical model for the simply-typed λ-calculus with $\beta\eta$-equality (properties *(B.i)-(B.iv)*) which, as well as inductive proof of β-equality, include the condition that currying and uncurrying are mutually inverse, *i.e.*, that products are left-adjoint to exponents (or equivalently, products/exponents are *categorical*, satisfying universal properties). Uustalu and Vene used the structure of a *monoidal comonad* which provides additional axioms that a monoidal functor preserves the comonad operations.

Definition 3.3.8. A *lax* or *colax* monoidal comonad has *lax/colax monoidal counit* and *co-multiplication* natural transformations, *i.e.*, the following diagrams commute in corresponding directions for the lax or colax operations:

$$\begin{array}{ccc} \mathsf{D}A \times \mathsf{D}B & \underset{\mathsf{n}_{A,B}}{\overset{\mathsf{m}_{A,B}}{\rightleftarrows}} & \mathsf{D}(A \times B) \\ \downarrow{\scriptstyle \delta_A \times \delta_B} & & \downarrow{\scriptstyle \delta_{A\times B}} \\ \mathsf{DD}A \times \mathsf{DD}B & \underset{\mathsf{n}_{\mathsf{D}A,\mathsf{D}B}}{\overset{\mathsf{m}_{\mathsf{D}A,\mathsf{D}B}}{\rightleftarrows}} \mathsf{D}(\mathsf{D}A \times \mathsf{D}B) & \underset{\mathsf{D}\mathsf{n}_{A,B}}{\overset{\mathsf{D}\mathsf{m}_{A,B}}{\rightleftarrows}} \mathsf{DD}(A \times B) \end{array} \qquad \begin{array}{ccc} \mathsf{D}A \times \mathsf{D}B & \underset{\mathsf{n}_{A,B}}{\overset{\mathsf{m}_{A,B}}{\rightleftarrows}} & \mathsf{D}(A \times B) \\ \downarrow{\scriptstyle \varepsilon_A \times \varepsilon_B} & & \downarrow{\scriptstyle \varepsilon_{A\times B}} \\ A \times B & = & A \times B \end{array} \tag{33}$$

$$\begin{array}{ccc} I & \underset{\mathsf{n}_1}{\overset{\mathsf{m}_1}{\rightleftarrows}} & \mathsf{D}I \\ \| & & \downarrow{\scriptstyle \delta_I} \\ I & \underset{\mathsf{n}_1}{\overset{\mathsf{m}_1}{\rightleftarrows}} \mathsf{D}I & \underset{\mathsf{D}\mathsf{n}_1}{\overset{\mathsf{D}\mathsf{m}_1}{\rightleftarrows}} \mathsf{DD}I \end{array} \qquad \begin{array}{ccc} I & \underset{\mathsf{n}_1}{\overset{\mathsf{m}_1}{\rightleftarrows}} & \mathsf{D}I \\ & \searrow & \downarrow{\scriptstyle \varepsilon_1} \\ & & I \end{array} \tag{34}$$

If n_1 and m_1, or $\mathsf{n}_{A,B}$ and $\mathsf{m}_{A,B}$, are isomorphisms then a proof of any of the above diagrams commuting in one direction implies the dual property for the dual operation.

The rest of this section expands the work of Uustalu and Vene by exploring the equational theory of the contextual λ-calculus and the corresponding required conditions. This section shows the use of the monoidal comonad axioms, following the scheme in Section 2.2.7 (p. 33).

***Let*-binding = application-abstraction pair.** Recall that the [≡-*let*-λ] property, *i.e.*, $(\lambda x.e_1)e_2 \equiv \textbf{let}\ x = e_2\ \textbf{in}\ e_1$, requires $\Psi \circ \Phi = id$ (Section 2.2.7, (*(B.i)*)), which simplifies as follows:

$$\begin{aligned}\Psi \circ \Phi &= \mathbb{C}(\mathsf{n}_{A,X}, -) \circ \phi^{-1}_{A,B} \circ \phi_{A,B} \circ \mathbb{C}(\mathsf{m}_{A,X}, -) \\ &= \mathbb{C}(\mathsf{n}_{A,X}, -) \circ \mathbb{C}(\mathsf{m}_{A,X}, -) && \{\textit{by underlying adjunction } \phi^{-1} \circ \phi = id\} \\ &= \mathbb{C}(\mathsf{m}_{A,X} \circ \mathsf{n}_{A,X}, -) && \{\mathbb{C}(-,-) : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbb{C}\ \textit{functoriality}\}\end{aligned}$$

Therefore, $\Psi \circ \Phi = id$ requires that $\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id$. An intermediate case arises, when $\mathsf{m}_{A,B}$ is idempotent lax and $\mathsf{n}_{A,B}$ is taken as the canonical colax structure for F:

Proposition 3.3.9. *For a lax (semi-)monoidal endofunctor* F *on a category with finite products, and canonical colax operation* $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$ *the following two properties are equivalent: (for which the proof is given in Appendix C.2.1 (p. 197))*

(i) $\mathsf{F}\Delta = \mathsf{m}_{A,B} \circ \Delta$ *(i.e.* F *is idempotent lax monoidal);*
(ii) $\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id$.

Therefore, for the canonical $\mathsf{n}_{A,B}$, [≡-*let*-λ] requires that D is lax/colax (semi)monoidal with $\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id$, or $\mathsf{m}_{A,B}$ is idempotent. Uustalu and Vene note that idempotency[4] is "automatic" if the functor is strong symmetric monoidal [**UV08**]. Proposition 3.3.9 shows a slightly more general result when $\mathsf{m}_{A,B}$ is just right-inverse to $\mathsf{m}_{A,B}$, rather than the stronger property of $\mathsf{m}_{A,B}$ and $\mathsf{n}_{A,B}$ as mutually inverse.

η-equality. For $\lambda x.f\,x \equiv f$ to hold then $\Phi \circ \Psi = id$ (Section 2.2.7, (*(B.iii)*)). Expanding $\Phi \circ \Psi$ and simplifying gives:

$$\begin{aligned}\Phi \circ \Psi &= \phi_{A,B} \circ \mathbb{C}(\mathsf{m}_{A,X}, -) \circ \mathbb{C}(\mathsf{n}_{A,X}, -) \circ \phi^{-1}_{A,B} \\ &= \phi_{A,B} \circ \mathbb{C}(\mathsf{n}_{A,X} \circ \mathsf{m}_{A,X}, -) \circ \phi^{-1}_{A,B} && \{\mathbb{C}(-,-) : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbb{C}\ \textit{functoriality}\}\end{aligned}$$

Therefore, for Ψ to be the right-inverse of Φ then $\mathsf{n}_{A,X} \circ \mathsf{m}_{A,X} = id$. Furthermore, for η-equality one of the following must hold (Section 2.2.7, (*(B.iii)*)):

$$\Phi\,(g \circ^{\mathsf{D}} f) = (\Rightarrow^{\mathsf{D}} g) \circ^{\mathsf{D}} \Phi f \qquad (\textit{where } f : \mathsf{D}(A \times B) \to X, g : \mathsf{D}X \to Y) \tag{35}$$

$$\Psi(g \circ^{\mathsf{D}} f) = (\Psi g) \circ^{\mathsf{D}} (f \times^{\mathsf{D}} id^{\mathsf{D}}) \qquad (\textit{where } f : \mathsf{D}A \to X, g : \mathsf{D}X \to (\mathsf{D}B \Rightarrow Y)) \tag{36}$$

Subsequently, either $\times^{\mathsf{D}}$ or $\Rightarrow^{\mathsf{D}}$ must be defined as a bifunctor (not just an object mapping). For (36), the bifunctor $(- \times^{\mathsf{D}} -) : \mathbb{C}_{\mathsf{D}} \times \mathbb{C}_{\mathsf{D}} \to \mathbb{C}_{\mathsf{D}}$ is defined for objects and morphisms as:

- $(A \times^{D} B) = A \times B \in \mathbb{C}_{\mathsf{D}0}$
- $(f \times^{D} g) = (f \times g) \circ \mathsf{n}_{A,B} : A \times^{D} B \to X \times^{D} Y \in \mathbb{C}_{\mathsf{D}1}$ (*i.e.*$f : \mathsf{D}A \to X, g : \mathsf{D}B \to Y \in \mathbb{C}_1$ and $(f \times^{D} g) : \mathsf{D}(A \times B) \to X \times Y \in \mathbb{C}_1$)

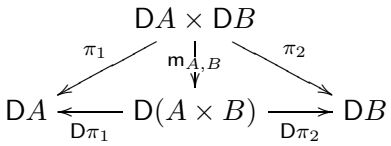
Proof of (36) is given in Appendix C.2.2 (p. 198), which requires D to be a *colax monoidal comonad*, requiring δ to be preserved by $\mathsf{n}_{A,B}$ (33).

[4] Uustalu and Vene do not call this property *idempotency*; this terminology is new here.

β-equality. If [$\equiv$-*let*-λ] holds, then β-equality can be proved by proving the following lemma that let-binding internalises syntactic substitution (*let*-β):

$$\forall e, e'.\ (\Gamma, x : \tau' \vdash e : \tau) \wedge (\Gamma \vdash e' : \tau') \Rightarrow [\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau]\!] = [\![\Gamma \vdash e[x := e'] : \tau]\!]$$

A proof, which proceeds by induction over the structure of type derivations, is shown in Appendix C.2.3 (p. 199). The proof requires that ε and δ are lax and colax semi-monoidal natural transformations, *i.e.*, D is a lax semi-monoidal comonad, and the additional properties:



These two properties are equivalent to requiring $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} = id$:

Proposition 3.3.10. *For a lax (semi-)monoidal endofunctor* F *on a category with finite products and canonical colax operation* $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$*, the following two properties are equivalent: (for which the proof is given in Appendix C.2.1 (p. 198)*

(i) $\mathsf{F}\pi_1 \circ \mathsf{m}_{A,B} = \pi_1$ *and* $\mathsf{F}\pi_2 \circ \mathsf{m}_{A,B} = \pi_2$*;*
(ii) $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} = id$*.*

Summary. Thus, the following syntactic properties of the contextual λ-calculus imply the following structure additional to a comonad D:

- [$\equiv$-*let*-λ] – $(\lambda x.e_1)\, e_2 \equiv \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2$ if either:
  - D is lax/colax (semi-)monoidal and $\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id$;
  - *or* D is lax/colax (semi-)monoidal, with canonical n and idempotent m (Proposition 3.3.9).
- [$\equiv$-η] – $\lambda x.f\, x \equiv f$ if:
  - D is a colax monoidal comonad and lax/colax (semi-)monoidal functor and $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} = id$.
- [$\equiv$-β] – $(\lambda x.e_1)\, e_2 \equiv e_1[x := e_2]$ if either:
  - D is a lax and colax monoidal comonad and $\mathsf{D}\pi_1 \circ \mathsf{m}_{A,B} = \pi_1$ and $\mathsf{D}\pi_2 \circ \mathsf{m}_{A,B} = \pi_2$;
  - *or* D is a lax and colax monoidal comonad and a lax/colax (semi-)monoidal functor with canonical n and $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} = id$ (Proposition 3.3.10).

Therefore, full $\beta\eta$-equality requires a *strong monoidal comonad*, equivalent to saying the coKleisli category of D is *Cartesian-closed* when the underlying comonad is strong monoidal.

3.3.3. Example: Lucid

As established in Section 3.2.4, only general streams comonads (that allow past and future elements to be accessed) can provide a semantics for Lucid which includes its *fby* and *next* operations. Since the stream zipper and pointed stream are isomorphic, the pointed stream defined by $\mathsf{InContext}\,\mathbb{N}$ is used here for ease of notation.

For a first-order Lucid, without function abstraction, only the additional structure of a colax monoidal functor is required (with $\mathsf{n}_{A,B}$), ignoring any equational theory for the moment.

For a higher-order Lucid with abstraction, the lax monoidal functor structure is required (with $\mathsf{m}_{A,B}$). Since this combines two-pointed streams, the cursors must be combined in some way. The following provides a suitable definition where the minimum of two cursor is taken:

$$\mathsf{m}\ (\mathit{Stream}\ s\ c, \mathit{Stream}\ t\ d) = \mathit{Stream}\ (\lambda c' \to (s\ (c' - (c\ \mathsf{min}\ d) + c), \\ t\ (c' - (c\ \mathsf{min}\ d) + d)))\ (c\ \mathsf{min}\ d)$$

The minimum is used instead of the maximum, which would produce negative cursors which are not allowed as the domain of the *in context* comonad is $\mathbb{N}$. The unit operation of a monoidal functor requires the operation $\mathsf{m}_1 : a \to \mathit{Stream}\, a$ here, which is the unit of m. This however requires extending the domain $\mathbb{N}$ to $\mathbb{N} \cup \{\omega\}$ where $\mathsf{m}_1\, x = \mathit{Stream}\ (\lambda c \to x)\, \omega$.

The above definition of $\mathsf{m}_{A,B}$ and the canonical $\mathsf{n}_{A,B}$ provides a lax/colax semi-monoidal comonad satisfying (33) where $\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id$ thus [$\equiv$-*let*-λ] holds. However, the dual property $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} = id$ does not hold since $\mathsf{m}_{A,B}$ takes the minimum of the cursors, which $\mathsf{n}_{A,B}$ cannot invert. Therefore [$\equiv$-η] and [$\equiv$-β] do not hold. However, since minimum is idempotent, if $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B}$ is applied to a pair of streams which have the same cursor then $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} \circ (f \times g) = (f \times g)$, where f and g are functions returning two streams with the same cursor. This implies some limited cases where $\beta\eta$-equality can hold, but not in general.

3.4. Monoidal comonads and shape

The notion of *shape* for a parametric type's values is defined generically by "erasing" elements.

Definition 3.4.1. For an endofunctor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$, where $\mathbb{C}$ has a terminal object $1 \in \mathbb{C}_0$, the shape of any object $\mathsf{F}\, A$ is determined by the natural transformation $\mathsf{shape}_A = \mathsf{F}\, !_A : \mathsf{F}A \to \mathsf{F}1$.

In a functional programming interpretation, the shape of a parametric data type that is functorial is thus computed by mapping all its elements to the (single-valued) unit type ().

The comonad axioms imply an important property of comonads which reveals much about their utility: that coKleisli-extended functions are *shape preserving*.

Lemma 3.4.2 (Shape preservation)**.** *Let* $(\mathsf{D}, \varepsilon, (-)^\dagger)$ *be a coKleisli triple, on a category* $\mathbb{C}$ *with a terminal object. Extension of any coKleisli morphism* $f : \mathsf{D}A \to B$ *yields a* shape preserving *morphism* $f^\dagger : \mathsf{D}A \to \mathsf{D}B$ *i.e. the shape of the parameter object is preserved in the result thus:*

$$\mathsf{D}\, !_B \circ f^\dagger = \mathsf{D}\, !_A \tag{37}$$

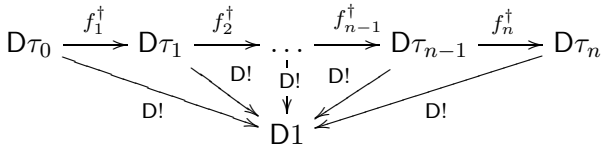
Proof. Recall the *terminal property* that morphisms to the terminal object of a category 1 are unique; *shape preservation* is thus proved:

$$\begin{array}{lll} \mathsf{D}\, ! \circ f^\dagger = & \mathsf{D}\, ! \circ \mathsf{D}f \circ \delta & \{f^\dagger = \mathsf{D}f \circ \delta\} \\ = & \mathsf{D}(! \circ f) \circ \delta & \{\mathsf{D}\ \mathit{functoriality}\} \\ = & \mathsf{D}(! \circ \varepsilon) \circ \delta & \{!_X \circ f = !_A = !_Y \circ g \quad \mathit{universal\ property}\} \\ = & \mathsf{D}\, ! \circ \mathsf{D}\varepsilon \circ \delta & \{\mathsf{D}\ \mathit{functoriality}\} \\ = & \mathsf{D}\, ! & [\mathrm{C2}] \end{array}$$

□

The crucial step in the proof is the use of the [C2] axiom (equivalent to [coK2]), which enforces preservation of shape for the comonad.

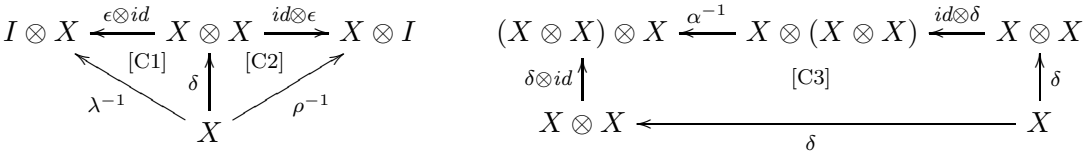
This contrasts with monads where the shape is permitted to change, that is, monads encapsulate effects which may change the execution of monadic computations that are post-composed to some monadic computation. Whilst comonads are the dual of monads, the execution of pre-composed computations is not changed by a computation. Instead, execution for composed coKleisli morphisms is uniform throughout by shape preservation. Thus, for the composition of morphisms $f_1 : \mathsf{D}\tau_0 \to \tau_1, \ldots, f_n : \mathsf{D}\tau_{n-1} \to \tau_n$ the following commutes by shape preservation:



(38)

The asymmetry between monads and comonads here is analogous to that between monoids and comonoids in **Set** and CCCs. Indeed, monads and comonads *are* monoids and comonoids over the monoidal category of endofunctors $([\mathbb{C},\mathbb{C}], \circ, 1_{\mathbb{C}})$. Comonoids are defined as follows:

Definition 3.4.3. A *comonoid*, on a monoidal category $(\mathbb{C}, \otimes, I)$, has an object X with comultiplication $\delta : X \otimes X \to X$ and counit $\epsilon : X \to I$, satisfying coassociativity and counitality:



This definition dualises that of monoids in a monoidal category (see Appendix B.1.4, p. 186).

Thus, for the category of endofunctors $([\mathbb{C},\mathbb{C}], \circ, 1_{\mathbb{C}})$, the axioms of a comonad imply that the encoded contextual information is decomposed *comonoidally*. Whilst there are many examples of monoids in **Set** and CCCs, there is only one comonoid: the trivial *duplication comonoid*, given by the diagonal $\Delta_A : A \to A \times A$ and terminal morphisms $!_A : A \to 1$. Thus monoids and comonoids seem asymmetric in their utility in a CCC. For comonads, the comonoid's uniformity manifests itself as the shape preservation property.

In principle, the product comonad $\mathsf{D}A = A \times X$ can be generalised given a comonoid structure $(X, (\boldsymbol{\delta} : X \to X \times X), (\boldsymbol{\varepsilon} : X \to 1))$, with the comonads comultiplication defined:

$$\delta\ (a, x) = ((a, (\pi_1 \circ \boldsymbol{\delta})\ x), (\pi_2 \circ \boldsymbol{\delta})\ x)$$

However, in a CCC this general product comonad collapses to the original product comonad defined in Example 3.1.2 since the only possible comonoid is the trivial duplication comonoid.

3.4.1. Shape and semantics

The semantics of Section 3.3 required a lax and colax monoidal functor, with additional properties for various equational theories. It was shown that $\beta\eta$-equality holds for colax/lax monoidal comonads with the canonical colax monoidal operation $\mathsf{n}_{A,B} = \langle \mathsf{D}\pi_1, \mathsf{D}\pi_2 \rangle$ if $\mathsf{m}_{A,B}$ is idempotent and $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} = id$. For Lucid, with a general stream comonad, it was shown in Section 3.3.3 that this property does not, and cannot, in general hold. However, it was noted that this property holds if $\mathsf{m}_{A,B}$ is applied to two streams with the same cursor. This property can be generalised for any comonad where $\mathsf{m}_{A,B}$ is passed two values of the same shape.

Lemma 3.4.4. *For an idempotent monoidal functor* F*, the canonical colax monoidal operation* $\mathsf{n}_{A,B} = \langle \mathsf{D}\pi_1, \mathsf{D}\pi_2 \rangle$ *is the left-inverse of the monoidal operation* $\mathsf{m}_{A,B}$ *when pre-composed with morphisms whose images are of equal shape, i.e., for* $f : A \to \mathsf{F}X, g : B \to \mathsf{F}Y$*, then:*

$$\mathsf{shape}_X \circ f \circ \pi_1 = \mathsf{shape}_Y \circ g \circ \pi_2 \quad \Rightarrow \quad \mathsf{n}_{X,Y} \circ \mathsf{m}_{X,Y} \circ (f \times g) = f \times g \tag{39}$$

The proof is shown in Appendix C.2.4 (p. 202) which uses the universal properties of products and $\mathsf{m}_{A,B}$ idempotency (as specified in the lemma).

Corollary 3.4.5. *The colax monoidality witness* $\mathsf{n}_{A,B} = \langle \mathsf{D}\pi_1, \mathsf{D}\pi_2 \rangle$ *is shape preserving:*

$$\begin{array}{ccc} \mathsf{D}(A \times B) & \xrightarrow{\mathsf{n}_{A,B}} & \mathsf{D}A \times \mathsf{D}B \\ {\scriptstyle \mathsf{shape}_{A\times B}}\downarrow & & \downarrow{\scriptstyle \mathsf{shape}_A \times \mathsf{shape}_B} \\ \mathsf{D}1 & \xrightarrow[\Delta]{} & \mathsf{D}1 \times \mathsf{D}1 \end{array}$$

Lemma 39 and Corollary 3.4.5 together show that, in the case where $\mathsf{m}_{A,B}$ is idempotent, the core semantics of the general contextual λ-calculus has contexts of uniform shape throughout. This restricts the computations that can be structured by (monoidal) comonads to those with uniform contextual requirements. Chapter 5 generalises comonads, relaxing this uniformity.

Contexts of non-uniform shape may be introduced by instances of the contextual calculus for specific notions of context. For example, a particular notion of contextual computation may have built-in constructs that introduces contexts of a different shape to those being handled by the core semantics. For example, consider a construct with denotation $f : \mathsf{D}(\Gamma \times (\mathsf{D}A \to B)) \to B$ defined $f = \lambda x.(\pi_2\,(\varepsilon\,x))\,c$ where $c : \mathsf{D}A$ is some constant context built into the semantics whose shape differs to that of the incoming context x.

3.4.2. Containers and shape

Many of the data structure considered in this chapter are *containers*, that is, parametric data types which have "holes" filled by data of the parameter type. The general concept of a container has been formalised by Abbott, Altenkirch, and Ghani, by decomposing container types into a *shape* (or *template*) and a mapping from *positions* (or *holes*) in the shape to the contained values [**AAMG04**, **AAG05**]. For example, the list type $[A]$ is a container whose possible shapes

are natural numbers $n : \mathbb{N}$ representing the length of a list, thus positions are elements of the set $\{0, \ldots, n-1\}$, and the values of a list of length n are described by a map $\{0, \ldots, n-1\} \to A$.

Containers are described generally by a dependent sum on their shapes:

$$(S \lhd P)A = \sum_{s:S} (Ps \to A)$$

where $(S \lhd P)$ denotes the container with set S of shapes and indexed family P of positions, encoded by a coproduct of maps from positions to values for each possible shape.

Examples 3.4.6 A few examples are:

- the (non-empty) list container $(\mathbb{N}_{>0} \lhd \mathsf{Fin})$ where $\mathsf{Fin}\ n = \{0, \ldots, n-1\}$
- the stream container $(\{*\} \lhd \mathit{const}\ \mathbb{N})$, *i.e.*, there is only one shape since streams are infinite, where *const* is the constant indexed family.
- numerical functions are containers with $(\{*\} \lhd \mathit{const}\, \mathbb{R})$.
- finite n-dimensional arrays with container $(\mathbb{N}^n_{>0} \lhd \mathsf{Fin}^n)$

Ahman *et al.* recently showed that all *directed containers* – those with notions of *sub-shape* – are comonads, where positions are contexts and sub-shapes define accessibility between contexts for the definition of *extend* [**ACU12**]. For example, suffix trees, lists, and streams, are directed, with subshapes (*i.e.* sublists, subtrees) provided by a monoidal structure on their cursor.

General *poly*shape container transformations have type $f : \sum_{s \in S}(Ps \to A) \to \sum_{s \in S}(Ps \to B)$, thus a container of different shape can be returned. However, shape preserving container transformations have type:

$$\sum_{s \in S}((Ps \to A) \to (Ps \to B))$$

where a container of the same shape is returned. The *in context* comonad can be generalised to a cursored-container comonad with the type $\mathsf{D}\, A = \sum_{s \in S}(Ps \to A) \times Ps$.

Monoshape containers, *i.e.*, with a single shape such as streams, have a comonad provided by the *in context* comonad with positions as the contexts. A general function on monoshape containers with shape s, $f : (P\, s \to A) \to (P\, s \to B)$ is therefore a coKleisli morphism of the *in context* comonad by *uncurry* $f : (P\, s \to A) \times P\, s \to B$.

Proposition 3.4.7. *Any monoshape container* $(\{s\} \lhd P)$ *is a strong monoidal functor.*

Proof. Monoshape containers are equivalent to exponents where $(\{s\} \lhd \mathit{const}\, P)A = P \Rightarrow A$, with canonical colax operation $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$ and lax operation $\mathsf{m}_{A,B}(x, y) = \langle x, y \rangle$. □

Uustalu and Vene define a causal dataflow comonad on $[\mathbb{N}, \mathbf{Set}]$ where list values are indexed by their length $(\mathsf{D}A)_n$, which they call *precise comonads* [**UV08**]. They show that these monoshape containers are strong monoidal comonads with monoidal operations $(\mathsf{m}_{A,B})_n : (\mathsf{D}A)_n \times (\mathsf{D}B)_n \to (\mathsf{D}(A \times B))_\mathsf{n}$. Chapter 5 and 6 generalise comonads using indexed families, but with operations such as $(\mathsf{m}_{A,B})_{n,m} = \mathsf{D}_n A \times \mathsf{D}_m B \to \mathsf{D}_{n \sqcap m}(A \times B)$ where operations on indices describe how different contexts are composed and combined.

3.5. Relating monads and comonads

Both monads and comonads can be used to encapsulate *impure* features of a computation. Comonads encapsulate input-related impurity (*e.g.*, demands on the context); monads encapsulate output-related impurity (*e.g.* (global) changes to the context), traditionally called *effects*.

An interesting case arises with the notion of *parameterised computations*, which can be structured by either a monad or comonad. As shown earlier, the product comonad, where $\mathsf{D}A = A \times X$, structures computations with implicit parameters. The *reader monad*, with functor $\mathsf{M}A = X \Rightarrow A$, is also used in functional programming for abstracting implicit parameters as an effect [**Hug04**, **Wad95b**]. It is not surprising that the exponent monad and product comonad encode the same notion of impurity since their Kleisli and coKleisli morphisms are isomorphic via currying/uncurrying, from the adjunction $(- \times X) \dashv (X \Rightarrow -)$ (Example 2.2.13, p. 31):

$$(A \times X) \to B \underset{\text{uncurry}}{\overset{\text{curry}}{\rightleftarrows}} A \to (X \to B)$$

This isomorphism preserves composition between the corresponding Kleisli and coKleisli categories such that the two categories are isomorphic, showing that the product comonad and exponent monad are equivalent in their ability to structure computation, or *equipotent*.

Definition 3.5.1. A monad with endofunctor $\mathsf{M} : \mathbb{C} \to \mathbb{C}$ and a comonad with $\mathsf{D} : \mathbb{C} \to \mathbb{C}$ are *equipotent* if their corresponding (co)Kleisli categories are isomorphic $\mathbb{C}_{\mathsf{M}} \cong {}_{\mathsf{D}}\mathbb{C}$, *i.e.*, there are functors $\Phi : \mathbb{C}_{\mathsf{M}} \to {}_{\mathsf{D}}\mathbb{C}$, $\Phi^{-1} : {}_{\mathsf{D}}\mathbb{C} \to \mathbb{C}_{\mathsf{M}}$ witnessing an isomorphism.

Thus, equipotency is defined by an isomorphism between the semantic domains of an effectful and contextual language, where the functors witnessing the isomorphism exchange impurity in the input of a computation for impurity in the output. For the product comonad and exponent monad, currying/uncurrying provides this, where the functorial laws are easily proved.

Previously, Eilenberg and Moore showed that given a monad M, if there exists an endofunctor R such that $\mathsf{M} \dashv \mathsf{R}$ then R has a comonad structure [**EM65**]. Their result dualises providing an automatic equipotent structure:

Lemma 3.5.2. *Given a comonad* $\mathsf{D} : \mathbb{C} \to \mathbb{C}$*, if* D *has a right adjoint* R*, i.e.,* $\mathsf{D} \dashv \mathsf{R}$*, then* R *is a monad which is equipotent to* D*.*

The adjunction $\mathsf{D} \dashv \mathsf{R}$ provides the isomorphism $\mathbb{C}(\mathsf{D}A, B) \cong \mathbb{C}(A, \mathsf{R}B)$ which leads to equipotence of D and R. The proof of this lemma is not shown here as the result does not appear to be particularly useful in the Cartesian-closed setting, where the product comonad and exponent monad appear to be the only example.

A related form of *weak* equipotence provides a resolution to an interesting tension in the literature on the categorical semantics of Lucid.

Weak equipotence. In 1995, Wadge proposed that the semantics of Lucid could be structured monadically [**Wad95a**]. Ten years later, Uustalu and Vene gave a comonadic semantics for Lucid, and stated that "notions of dataflow cannot be structured with monads" [**UV06**]. There

is an apparent conflict which raises a number of questions: are the two approaches equivalent? What does "cannot be structured" mean? Is one approach in some sense better?

Wadge defines the semantics of Lucid using the exponent monad with a natural number domain: $WA = \mathbb{N} \Rightarrow A$, providing the intuition that W models streams, multiplication μ returns the *diagonal* of a *double-indexed* stream, and the unit η constructs the constant stream. The pointwise operations of Lucid are defined by lifting with η, e.g., $[\![+]\!] = \lambda x.\lambda y.\eta(x + y)$. However, Wadge does not address the monadic semantics of *intensional operators* (*e.g.*, *next* and *fby*). Whilst pointwise operators compute the n^{th} element of a stream from the n^{th} elements of its parameters, i.e., *extensionally*, an intensional operator may compute the n^{th} value of a stream from any value in a parameter stream. Thus, an intensional operator cannot be defined as a Kleisli morphism $A \to WB$, which takes just a single element, but must be a morphism $WA \to WB$. For example, *next* on W is defined:

$$next\ s = \lambda n.\, s\ (n + 1) : WA \to WA$$

Thus, the domain of Wadge's semantics cannot be the Kleisli category $\mathbb{C}_W$ but instead must be the category of morphisms $WA \to WB$ (or some suitable subcategory). The exponent monad therefore *weakly structures* Lucid's stream computations, with a mix of Kleisli morphisms and non-Kleisli morphisms of the form $WA \to WB$ which do not factor into Kleisli morphisms.

This deficiency of the exponent monad to capture intensional operators has been also noted in natural language semantics by Bekki, where the state monad was used instead [**Bek09**].

Uustalu and Vene's semantics, using the *in-context* stream comonad $\mathsf{Stream}A = (\mathbb{N} \Rightarrow A) \times \mathbb{N}$, *strongly structures* the semantics of Lucid where all its operations are coKleisli morphisms. Therefore, whilst Lucid *can* be technically structured by a monad, this structuring is *weak*, which has a number of disadvantages. Inevitably, the weak approach provides less *abstraction*. Subsequently, the general reasoning principles provided by a monad cannot be ubiquitously applied, since the non-Kleisli morphisms do adhere to these axioms. From a programming perspective, less abstraction decreases program modularity and increases the programming effort, providing more opportunity for errors. Furthermore, morphisms $WA \to WB$ do not interact well with specialised syntax, such as the **do**-notation, which is designed for Kleisli morphisms.

The semantics of Wadge and Uustalu-Vene are however equivalent by $(- \times X) \dashv (X \Rightarrow -)$:

$$(\mathbb{N} \Rightarrow A) \times \mathbb{N} \to B \underset{\text{uncurry}}{\overset{\text{curry}}{\rightleftarrows}} (\mathbb{N} \Rightarrow A) \to (\mathbb{N} \Rightarrow B)$$

The image of the endofunctor $WA = \mathbb{N} \Rightarrow A$ (*i.e.*, the subcategory with objects WA and morphisms $WA \to WB$) is isomorphic to the coKleisli category for $\mathsf{D}A = (\mathbb{N} \Rightarrow A) \times \mathbb{N}$. This result generalises as follows:

Lemma 3.5.3. *For a monad* M*, if* M *has a left adjoint* L*, i.e.,* $\mathsf{L} \dashv \mathsf{M}$*, then the comonad* LM *induced by the adjunction has a coKleisli category isomorphic to the image of* M*.*

This lemma dualises to a comonad D with a right adjoint $\mathsf{D} \dashv \mathsf{R}$. For example, the product/exponent adjunction shows that a semantics weakly structured by a product comonad with

morphisms $(A \times X) \to (B \times X)$ is equivalent to a strong structuring with the state monad, with morphisms $A \to (X \Rightarrow B \times X)$. For brevity, this general result is not discussed at length here nor proven. Furthermore, as with equipotency, the only useful examples in CCCs arise from the product/exponent adjunction. A technical report provides the derivations and proofs [**Orc12**].

Summary. The product/exponent adjunction provides an equivalence between the product comonad and exponent monad. Furthermore, this adjunction gives rise to equivalences between weak structuring with an exponent monad and strong structuring with the *in context* comonad, and weak structuring with product comonad and strong structuring with the state monad. Thus, both the Wadge and Uustalu-Vene approach are equivalent, although, as described, the comonadic approach has some advantages. In a CCC, there do not appear to be any other adjunctions (between endofunctors) which provides further useful relationships between monads and comonads (the identity monad and comonad are trivially equipotent).

3.6. Conclusion

This chapter introduced comonads formally and showed their use in programming. Various examples were shown. This chapter contributed a systematic derivation of a comonadic semantics based on the approach of Chapter 2, which generalised the work of Uustalu and Vene [**UV08**], making precise the additional structure required on top of a comonad for $\beta\eta$-equality in the contextual λ-calculus.

The asymmetry between monads and comonads was shown, in that effects encoded by monads can change the course of execution, whilst comonads have a uniform structure, where the shape of the context is preserved throughout. This property was elucidated by the *shape preservation lemma* (Lemma 3.4.2, p. 71).

Whilst monads are ubiquitous in functional programming, comonads are relatively underused. Shape preservation partly explains this underuse, since many computations on data structures change the shape, modifying the structure not just the values. Chapter 5 generalises comonads in a number of ways, including relaxation of shape preservation, widening the class of possible contextual computations and the applicability of a comonad-like approach.

The semantics here did not consider recursion, which is especially important for the semantics of Lucid. Uustalu and Vene previously showed how to include a semantics for recursion in Lucid [**UV06**]. Recursion/fixed-points are discussed again briefly in Section 8.2.1 (p. 169), but not considered further here.

The next chapter builds on the categorical semantic of contextual languages, developing notations (syntax) for simplifying programming of contextual computations that are embedded in Haskell, but could also form the basis for other embedded or non-embedded languages. The next chapter thus furthers the categorical programming approach for contextual computations.

Chapter 4

A notation for comonads

The previous chapter introduced the use of comonads in contextual semantics and in programming. Whilst languages such as Haskell and F# provide syntactic sugar for simplifying programming with monads (the **do**-notation and *let!* notation respectively [**PS12**]) there is no analogous notation for programming with comonads. This lack of language support contributes to the relative underuse of comonads as a programming idiom in these languages. This chapter contributes a notation which simplifies programming with comonads in Haskell and acts as an internal sublanguage for contextual programming in Haskell. The notation is used in Chapter 7 within a language for contextual computations on containers.

This chapter is based on the paper *A Notation for Comonads* by the author, published in the post proceedings of *IFL 2012* (*Symposium on Implementation and Application of Functional Languages*) [**OM12b**].

Chapter structure and contributions. The primary contribution of this chapter is the **codo**-notation[1], introduced in detail in Section 4.1. The notation desugars into the operations of a comonad (Section 4.2), thus an equational theory for the notation follows from the comonad laws (Section 4.3). The **codo**-notation is analogous to the **do**-notation for programming with monads, but with some notable differences which will be explained from a categorical semantics perspective in Section 4.4. Section 4.6 concludes, and discusses related work, including a comparison of the **codo**-notation to Haskell's *arrow notation*. A number of practical programming examples are included using the **codo**-notation, for example, the live-variable analysis from the introduction is shown using a control-flow graph comonad (Section 4.5).

4.1. Introducing the codo-notation

The *let!* and **do**-notation for programming with monads in F# and Haskell essentially provide internal sublanguages for effectful programming. Analogously, the **codo**-notation provides an internal sublanguage for contextual programming in Haskell.

As an initial example, the following **codo**-block composes local operations on arrays, defining an operation for computing the contours of a 2D-image using a *difference of Gaussians* approach:

$$
\begin{aligned}
&\mathit{contours} :: \mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float} \to \mathit{Float}\\
&\mathit{contours} = \mathbf{codo}\ x \Rightarrow y \leftarrow \mathit{gauss2D}\ x\\
&\qquad\qquad\qquad\quad\ z \leftarrow \mathit{gauss2D}\ y\\
&\qquad\qquad\qquad\quad\ w \leftarrow (\mathit{extract}\ y) - (\mathit{extract}\ z)\\
&\qquad\qquad\qquad\quad\ \mathit{laplace2D}\ w
\end{aligned}
$$

[1] The '**od**-notation' was considered as a name for this notation, but it was thought rather odd.

where *CArray i a* is the pointed array from Example 3.1.10 (p. 50), with index type i and element type a, and $gauss2D, laplace2D :: CArray\ (Int, Int)\ Float \to Float$ compute, at a particular context, the *discrete Gaussian* and *Laplace* operators. A contour image can thus be computed by applying (*extend contours*) to an image. The array comonad is used here to introduce **codo**.

The **codo**-notation provides a form of comonadic *let*-binding, with general form and type:

$$(\mathbf{codo}\ p \Rightarrow \overline{p \leftarrow e;}\, e) :: Comonad\ c \Rightarrow c\ t \to t'$$

(where p ranges over patterns, e over expressions, and t,t' over types). Compare this with the general form and type of the monadic **do**-notation:

$$(\mathbf{do}\ \overline{p \leftarrow e;} e) :: Monad\ m \Rightarrow m\ t$$

Both comprise zero or more binding statements of the form $p \leftarrow e$ (separated by semicolons or new lines), preceding a final result expression. A **codo**-block however defines a function, with a pattern-match on its parameter following the **codo** keyword. The parameter is essential since comonads provide composition for functions with structured input. A **do**-block is instead an expression (nullary function). Section 4.4 compares the two notations in detail.

codo for composition. The **codo**-notation abstracts *extend* in the composition of local operations. For example, composition of two array operations $lapGauss = laplace2D \circ (extend\ gauss2D)$ can be written equivalently in the **codo**-notation in a pointed-styled without *extend*:

$$\begin{aligned} lapGauss = \mathbf{codo}\ x \Rightarrow\ & y \leftarrow gauss2D\ x \\ & laplace2D\ y \end{aligned}$$

where $lapGauss :: CArray\ (Int, Int)\ Float \to Float$, $x, y :: CArray\ (Int, Int)\ Float$.

The parameter of a **codo**-block provides the context of the whole block where all subsequent local variables have the same context, following from *shape preservation*, *i.e.* x and y in the above example block are arrays of the same size with the same cursor.

For a variable-pattern parameter, a **codo**-block is typed by the following rule:

$$[\text{varP}]\frac{\Gamma; x : c\ t \vdash_c e : t'}{\Gamma \vdash \mathbf{codo}\ x \Rightarrow e : Comonad\ c \Rightarrow c\ t \to t'}$$

where $\vdash_c$ types statements of a **codo**-block. Judgments $\Gamma; \Delta \vdash_c \ldots$ have two sequences of variable-type assumptions: Γ for variables outside a block and Δ for variables local to a block. For example, variable-pattern statements are typed:

$$[\text{varB}]\frac{\Gamma; \Delta \vdash_c e : t \qquad \Gamma; \Delta, x : c\ t \vdash_c r : t'}{\Gamma; \Delta \vdash_c x \leftarrow e; r : t'}$$

where r ranges over the remaining statements and result expression, *i.e.*, $r = \overline{p \leftarrow e}; e'$.

The binding statements of a **codo**-block are equivalent to *let*-binding in the contextual λ-calculus of the previous chapter (Section 3.3), where $x \leftarrow e; r \cong \mathbf{let}\ x = e\ \mathbf{in}\ r$. The desugaring of **codo** follows a similar approach to the categorical semantics of *let* for the contextual λ-calculus. Informally, $(\mathbf{codo}\ y \Rightarrow x \leftarrow e; e')$ is desugared into $(\lambda x \to e') \circ (extend\ (\lambda y \to e))$.

Further typing rules for the **codo**-notation are collected in **Figure** 4.1.

$$\boxed{\Gamma \vdash \textbf{codo}\ p \Rightarrow e : \mathit{Comonad}\ c \Rightarrow c\ t \to t'}$$

$$[\text{varP}]\frac{\Gamma;\ x : c\ t \vdash_c e : t'}{\Gamma \vdash \textbf{codo}\ x \Rightarrow e : c\ t \to t'} \qquad [\text{tupP}]\frac{\Gamma;\ x : c\ t,\ y : c\ t' \vdash_c e : t''}{\Gamma \vdash \textbf{codo}\ (x, y) \Rightarrow e : c\ (t, t') \to t''}$$

$$[\text{wildP}]\frac{\Gamma; \emptyset \vdash_c e : t}{\Gamma \vdash \textbf{codo}\ \_ \Rightarrow e : \forall a.c\ a \to t}$$

$$\boxed{\Gamma; \Delta \vdash_c \overline{p \leftarrow e};\ e : t}$$

$$[\text{varB}]\frac{\Gamma; \Delta \vdash_c e : t \quad \Gamma; \Delta, x : c\ t \vdash_c r : t'}{\Gamma; \Delta \vdash_c x \leftarrow e;\ r : t'} \qquad [\text{tupB}]\frac{\Gamma; \Delta \vdash_c e : (t_1, t_2) \quad \Gamma; \Delta, x : c\ t_1, y : c\ t_2 \vdash_c r : t'}{\Gamma; \Delta \vdash_c (x, y) \leftarrow e;\ r : t'}$$

$$[\text{letB}]\frac{\Gamma; \Delta \vdash_c e : t \quad \Gamma; \Delta, x : t \vdash_c r : t'}{\Gamma; \Delta \vdash_c \textbf{let}\ x = e;\ r : t'} \qquad [\text{exp}]\frac{\Gamma \vdash e : t}{\Gamma; \cdot \vdash_c e : t} \qquad [\text{var}]\frac{\Gamma, v : c\ t; \Delta \vdash_c e : t'}{\Gamma; \Delta, v : c\ t \vdash_c e : t'}$$

Figure 4.1. Typing rules for the **codo**-notation

Non-linear plumbing. For the *lapGauss* example, **codo** does not provide a significant simplification. The **codo**-notation more clearly benefits computations which are not linear function composition (*e.g.*, those that use multi-parameter operations). Consider the following binary operation, which is polymorphic in the comonad:

$$\begin{aligned}&\mathit{minus} :: (\mathit{Comonad}\ c, \mathit{Num}\ a) \Rightarrow c\ a \to c\ a \to a\\ &\mathit{minus}\ x\ y = \mathit{current}\ x - \mathit{current}\ y\end{aligned}$$

This function subtracts its second parameter from its first at their respective current contexts. Using **codo**, *minus* can be used to compute *pointwise* subtraction, *e.g.*

$$\begin{aligned}\mathit{contours}' = \textbf{codo}\ x \Rightarrow\ & y \leftarrow \mathit{gauss2D}\ x\\ & z \leftarrow \mathit{gauss2D}\ y\\ & w \leftarrow \mathit{minus}\ y\ z\\ & \mathit{laplace2D}\ w\end{aligned}$$

(equivalent to *contours* in the introduction which inlined the definition of *minus*). The context, and therefore cursor, of every variable in the block is the same as that of x. Thus, y and z have the same cursor and *minus* is applied pointwise. The equivalent program without **codo** is considerably more complex:

$$\begin{aligned}\mathit{contours}'\ x = \textbf{let}\ & y = \mathit{extend}\ \mathit{gauss2D}\ x\\ & w = \mathit{extend}\ (\lambda y' \to \textbf{let}\ z = \mathit{extend}\ \mathit{gauss2D}\ y'\ \textbf{in}\ \mathit{minus}\ y'\ z)\ y\\ \textbf{in}\ & \mathit{laplace2D}\ w\end{aligned}$$

where the nested *extend* means that y' and z have the same cursor, thus $\mathit{minus}\ y'\ z$ is pointwise. An attempt at a simplification of this might be:

$$\begin{aligned} \mathit{contour\_bad}\ x = \mathbf{let}\ & y = \mathit{extend}\ \mathit{gauss2D}\ x \\ & z = \mathit{extend}\ \mathit{gauss2D}\ y \\ & w = \mathit{extend}\ (\mathit{minus}\ y)\ z \\ \mathbf{in}\ & \mathit{laplace2D}\ w \end{aligned}$$

However, $\mathit{extend}\ (\mathit{minus}\ y)\ z$ subtracts z at every context from y at a particular, fixed context, *i.e.*, this is not a pointwise subtraction. An equivalent expression to *contours_bad* can be written using nested **codo**-blocks:

$$\begin{aligned} \mathit{contour\_bad} = \mathbf{codo}\ x \Rightarrow\ & y \leftarrow \mathit{gauss2D}\ x \\ & (\mathbf{codo}\ y' \Rightarrow\ z \leftarrow \mathit{gauss2D}\ y' \\ & \qquad\qquad w \leftarrow \mathit{minus}\ y\ z \\ & \qquad\qquad \mathit{laplace2D}\ w)\ y \end{aligned}$$

where y in $\mathit{minus}\ y\ z$ is bound in the outer **codo**-block and thus has its cursor fixed, whilst z is bound in the inner **codo**-block and has its cursor varying. Variables bound outside of the nearest enclosing **codo**-block will be "unsynchronised" with respect to the context inside the block, *i.e.*, at a different context.

A **codo**-block may have multiple parameters in an uncurried-style, via tuple patterns ([tupP], **Figure** 4.1). For example, the following block has two parameters, which are Laplace-transformed and then pointwise added:

$$\begin{aligned} & \mathit{lapPlus} :: \mathit{CArray}\ \mathit{Int}\ (\mathit{Float}, \mathit{Float}) \rightarrow \mathit{Float} \\ & \mathit{lapPlus} = \mathbf{codo}\ (x, y) \Rightarrow\ a \leftarrow \mathit{laplace2D}\ x \\ & \qquad\qquad\qquad\qquad\quad b \leftarrow \mathit{laplace2D}\ y \\ & \qquad\qquad\qquad\qquad\quad (\mathit{current}\ a) + (\mathit{current}\ b) \end{aligned}$$

This has a single comonadic parameter with tuple elements, whose type is of the form $c\ (a, b)$. However, inside the block $x :: c\ a$ and $y :: c\ b$ as the desugaring of **codo** *unzips* the parameter (see Section 4.2). This can be contrasted with the denotations of the contextual λ-calculus where $[\![x : a, y : b \vdash e : \tau]\!] : \mathsf{D}(a \times b) \rightarrow \tau$. The comonadic tuple parameter ensures that multiple parameters have the same cursor. A pair of arguments to *lapPlus* must therefore be *zipped* first, provided by a lax monoidal functor operation, written here in Haskell as the *czip* operation:

$$\mathbf{class}\ \mathit{ComonadZip}\ c\ \mathbf{where}\ \mathit{czip} :: (c\ a, c\ b) \rightarrow c\ (a, b)$$

For *CArray*, *czip* can be defined:

$$\begin{aligned} & \mathbf{instance}\ (\mathit{Eq}\ i, \mathit{Ix}\ i) \Rightarrow \mathit{ComonadZip}\ (\mathit{CArray}\ i)\ \mathbf{where} \\ & \quad \mathit{czip}\ (\mathit{CA}\ a\ i, \mathit{CA}\ a'\ j) = \\ & \qquad \mathbf{if}\ (i \not\equiv j \vee \mathit{bounds}\ a \not\equiv \mathit{bounds}\ a')\ \mathbf{then}\ \mathit{error}\ \texttt{"Shape/cursor mismatch"} \\ & \qquad \mathbf{else\ let}\ es'' = \mathit{map}\ (\lambda k \rightarrow (k, (a\,!\,k, a'\,!\,k)))\ (\mathit{indices}\ a) \\ & \qquad\qquad \mathbf{in}\ \mathit{CA}\ (\mathit{array}\ (\mathit{bounds}\ a)\ es'')\ i \end{aligned}$$

Thus only arrays of the same shape can be zipped together, *i.e.*, *czip* is akin to a partial monoidal functor operation for monoshape comonadic values. In the contextual understanding, two parameter arrays are thus *synchronised* in their contexts. The example of *lapPlus* can therefore be applied to two (synchronised) array parameters x and y by *extend lapPlus* (*czip* (x, y)).

Arbitrary pattern matches can used for the parameter of a **codo**-block and on the left-hand side of binding statements. For example, the following uses a tuple pattern in a binding statement (see [tupB], **Figure** 4.1), which is equivalent to *lapPlus* by exchanging a parameter binding with a statement binding:

$$\begin{aligned} \mathit{lapPlus} = \mathbf{codo}\ z \Rightarrow\ & (x, y) \leftarrow \mathit{current}\ z \\ & a \leftarrow \mathit{laplace2D}\ x \\ & b \leftarrow \mathit{laplace2D}\ y \\ & \mathit{current}\ a + \mathit{current}\ b \end{aligned}$$

Tuple patterns were specifically discussed here since they provide a way of providing multiple parameters to a **codo**-block, as seen above. The typing of a general pattern in a statement, for some type constructor T, is roughly as follows:

$$[\text{patB}]\frac{\Gamma; \Delta \vdash_c e : T\,\bar{\tau} \qquad \Gamma; \Delta, \Delta' \vdash_c r : \tau' \qquad \mathit{dom}(\Delta') = \mathit{var\text{-}pats}(p)}{\Gamma; \Delta \vdash_c (T\,p) \leftarrow e;\ r : \tau'}$$

where $\mathit{dom}(\Delta')$ returns a set of the variables from a sequence of typing assumptions, and *var-pats* returns a set of the variable patterns nested in any pattern.

4.2. Desugaring codo

The desugaring of **codo** is based on the categorical semantics for the contextual λ-calculus in Section 3.3. Since, the **codo**-notation is just a *let*-binding syntax, the previous categorical semantics of application and abstraction, and thus the additional lax and colax monoidal structure, are not required. Therefore, the **codo**-notation is similar to the simple L_1 language described in Section 2.2.2 (p. 27) with just *let*-binding. Desugaring of **codo** is thus analogous to a categorical semantics of **codo** in a coKleisli category on **Hask** with products.

The desugaring is split into three parts for *binding statements*, *expressions* (on right-hand side of bindings), and the *outer block* syntax.

Bindings. Recall the categorical semantics of *let*-binding for the contextual λ-calculus:

$$\frac{[\![\Gamma \vdash e_1 : \sigma]\!] = k_1 : \mathsf{D}\Gamma \to \sigma \qquad [\![\Gamma, v : \sigma \vdash e_2 : \tau]\!] = k_2 : \mathsf{D}(\Gamma \times \sigma) \to \tau}{\begin{aligned}[\![\Gamma \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \tau]\!] &= k_2 \circ^{\mathsf{D}} \langle id, k_1 \rangle^{\mathsf{D}} : \mathsf{D}\Gamma \to \tau \\ &= k_2 \circ \langle \varepsilon, k_1 \rangle^{\dagger} : \mathsf{D}\Gamma \to \tau \end{aligned}}$$

The desugaring of a **codo**-block's bindings $[\![-]\!]$ (defined over $\vdash_c$) provides an analogous semantics into **Hask**, where morphisms are Haskell functions, with the following variable pattern rule:

$$\frac{[\![\Gamma; \Delta \vdash_c e : \sigma]\!] = k_1 : c\,\Delta \to \sigma \qquad [\![\Gamma; \Delta, v : \sigma \vdash_c r : \tau]\!] = k_2 : c\,(\Delta, \sigma) \to \tau}{[\![\Gamma; \Delta \vdash_c v \leftarrow e;\ r : \tau]\!] = k_2 \circ (\mathit{extend}\ (\lambda d \to (\mathit{current}\ d, k_1\ d))) : \mathit{Comonad}\ c \Rightarrow c\,\Delta \to \tau}$$

where Γ is the context of Haskell variables and Δ is the context of comonadic variables bound locally in the block. Note that Γ is never given an interpretation, *e.g.*, $[\![\Gamma; \Delta \vdash_c e : \sigma]\!] = k_1 : c\,\Delta \to \sigma$, since the usual semantics of Haskell for free variables is used; only the comonadic environment Δ is given an explicit interpretation in the generated functions. The interpretation $[\![\Gamma; \Delta \vdash_c e : \sigma]\!]$ (described shortly) desugars expressions that appear on the right-hand side of a statement or as the final expression of a block.

The local environment Δ is structured by left-associated tuples terminated by the unit type $()$. Thus, the desugaring of binding statements yields a Haskell function of type:

$$[\![\Gamma; v_1 : \sigma_1, \ldots, v_n : \sigma_n \vdash_c \overline{b} : \tau]\!] : \mathit{Comonad}\ c \Rightarrow c\,((((), \sigma_1), \ldots), \sigma_n) \to \tau$$

For tuple patterns the desugaring reassociates values from the tuple into the left-associated tuple format of the scope:

$$\frac{[\![\Gamma; \Delta \vdash_c e : (\sigma_1, \sigma_2)]\!] = k_1 : \mathsf{D}\Delta \to (\sigma_1, \sigma_2) \quad [\![\Gamma; \Delta, x : \sigma_1, y : \sigma_2 \vdash_c r : \tau]\!] = k_2 : \mathsf{D}((\Delta, \sigma_1), \sigma_2) \to \tau}{\begin{array}{l}[\![\Gamma; \Delta \vdash_c (x, y) \leftarrow e; r : \tau]\!] = k_2 \circ (\mathit{extend}\ (\lambda d \to (\lambda(\mathit{env}, (x, y)) \to ((\mathit{env}, x), y))\ (\mathit{current}\ d, k_1\ d))) \\ \qquad : \mathit{Comonad}\ c \Rightarrow c\,\Delta \to \tau\end{array}}$$

where $\lambda((\mathit{env}, (x, y)) \to ((\mathit{env}, x), y))$ reassociates the result into the left-nested tuple format of the environment; this desugaring generalises in the obvious way to arbitrary patterns.

Expressions. The desugaring of an expression unzips the incoming context, binding the values to the variables in Δ, with a local *let*-binding, into the scope of the expression:

$$\begin{array}{l}[\![v_1 : \sigma_1, \ldots, v_n : \sigma_n \vdash_c e : \tau]\!] = (\lambda\Delta \to \mathbf{let}\ [v_i = \mathit{fmap}\ (\mathit{snd} \circ \mathit{fst}^{n-i})\ \Delta]_1^n\ \mathbf{in}\ e) \\ \qquad : \mathit{Comonad}\ c \Rightarrow c\,((((), \sigma_1), \ldots), \sigma_n) \to \tau\end{array}$$

where fst^k means k compositions of fst and $\mathit{fst}^0 = \mathit{id}$. This is analogous to the usual categorical semantics of variable access (see equation (14), p. 28).

Outer block. The top-level of a **codo**-block provides the interface between the **codo**-notation and Haskell. For a **codo**-block with a variable pattern parameter, the desugaring is as follows:

$$\frac{[\![\Gamma; v : \sigma \vdash_c b : \tau]\!] = k : c\,((), \sigma) \to \tau}{[\![\Gamma \vdash \mathbf{codo}\ v \Rightarrow b : \tau]\!] = k \circ \mathit{fmap}\ (\lambda x \to ((), x)) : \mathit{Comonad}\ c \Rightarrow c\,\sigma \to \tau}$$

where $\mathit{fmap}\ (\lambda x \to ((), x))$ projects the incoming parameter comonad to the left-nested pair format of the blocks local scope. For tuple patterns, this desugaring is:

$$\frac{[\![\Gamma; x : \sigma_1, y : \sigma_2 \vdash_c b : \tau]\!] = k : c\,(((), \sigma_1), \sigma_2) \to \tau}{[\![\Gamma \vdash \mathbf{codo}\ (x, y) \Rightarrow b : \tau]\!] = k \circ \mathit{fmap}\ (\lambda p \to (\mathit{fst}\ p, (\mathit{snd}\ p, ()))) : \mathit{Comonad}\ c \Rightarrow c(\sigma_1, \sigma_2) \to \tau}$$

This generalises easily to arbitrary patterns. In each case, the interpretation of the bindings is pre-composed with a lifted projection function, which projects values inside the incoming parameter comonad to left-nested pairs terminated by $()$. For a wildcard pattern the desugaring is: $[\![\Gamma \vdash \mathbf{codo}\ \_ \Rightarrow b : \tau]\!] = [\![\Gamma; \emptyset \vdash_c b]\!] \circ (\mathit{fmap}\ (\lambda x \to (x, ()))) : \mathit{Comonad}\ c \Rightarrow c\ a \to \tau$.

Example. The following gives an example of the desugaring, for **codo** $x \Rightarrow y \leftarrow f\ x;\ \ h\ x\ y$:

$$\begin{aligned}&(\lambda\Delta \to \textbf{let}\ y = \mathit{fmap}\ \mathit{snd}\ \Delta\\ &\qquad\qquad\quad x = \mathit{fmap}\ (\mathit{snd} \circ \mathit{fst})\ \Delta\ \textbf{in}\ h\ x\ y)\\ &\circ (\mathit{extend}\ (\lambda\Delta \to (\textbf{let}\ x = \mathit{fmap}\ \mathit{snd}\ \Delta\ \textbf{in}\ (\mathit{current}\ \Delta, f\ x))))\\ &\circ (\mathit{fmap}\ (\lambda x \to ((), x)))\end{aligned}$$

Section 4.4 compares the **codo**-notation with **do**-notation, and explains why the desugaring of **codo**-notation is more complex.

4.2.1. Rewriting and optimisation

The axioms of category-theoretic structures used in programming can often be oriented as *rewrite rules* which provide program optimisations. For example, preservation of composition by a functor yields a rewrite rule $(\mathsf{F}g \circ \mathsf{F}f) \rightsquigarrow \mathsf{F}(g \circ f)$ which reduces two traversals of a data structure into one, eliminating an intermediate data structure (a *deforestation* [**Wad90a**]). Even more beneficially, the comonad axioms provide rewrite rules which change the asymptotic complexity of a program. Assuming a container comonad with $(\mathit{fmap}\ f)$, $(\mathit{extend}\ f) \in \mathcal{O}(n|f|)$ and $\mathit{current} \in \mathcal{O}(1)$, its axioms provide rewrite rules which make the following asymptotic improvements:[2]

ε naturality	$\mathit{current} \circ (\mathit{fmap}\ f)$	$\rightsquigarrow$	$f \circ \mathit{current}$	$\in \mathcal{O}(n\|f\|)$	$\rightsquigarrow$	$\in \mathcal{O}(\|f\|)$
[cK1]	$\mathit{extend}\ \mathit{current}$	$\rightsquigarrow$	id	$\in \mathcal{O}(n)$	$\rightsquigarrow$	$\in \mathcal{O}(1)$
[cK2]	$\mathit{current} \circ (\mathit{extend}\ f)$	$\rightsquigarrow$	f	$\in \mathcal{O}(n\|f\|)$	$\rightsquigarrow$	$\in \mathcal{O}(\|f\|)$
[cK3]	$\mathit{extend}\ (g \circ \mathit{extend}\ f)$	$\rightsquigarrow$	$\mathit{extend}\ g \circ \mathit{extend}\ f$	$\in \mathcal{O}(n\|g\| + n^2\|f\|)$	$\rightsquigarrow$	$\in \mathcal{O}(n\|g\| + n\|f\|)$

The ε naturality and [cK2] optimisations are largely redundant in a lazy language. However, [cK1] and [cK3] require either the programmer or the compiler to make the optimisation.

For a **codo**-block with n- statements, its desugaring yields a coKleisli morphism of the form:

$$m = h \circ (g_n^\dagger \circ \ldots \circ g_1^\dagger) \circ \mathsf{D}p \tag{40}$$

where h is the desugaring of the final result expression and p is the projection mapping the parameter of the block to the left-nested tuples format of the comonadic environment. The function m may be used within another **codo**-block, or it may be applied using *extend*. For a comonad with $(\mathit{extend}\ f) \in \mathcal{O}(n|f|)$ and $m' = \mathit{extend}\ m$ then $m' \in \mathcal{O}(n^2|f|)$ (assuming $h, p, g_i \in \mathcal{O}(1)$). However, the same semantics is provided if $m' = h^\dagger \circ (g_n^\dagger \circ \ldots \circ g_1^\dagger) \circ \mathsf{D}p$ but with a lower complexity $m' \in \mathcal{O}(n|f|)$.

Rewriting nested applications of *extension* is therefore crucial to the usability of **codo**-notation without performance loss. GHC/Haskell provides a mechanism for user-defined rewrite rules [**JTH01**], which can inform the compiler of this optimisation as follows:

[2] Note, some comonads have operations with worse asymptotic complexity. For example, the pointed-list comonad ($\mathsf{D}A = [A] \times \mathbb{N}$), has $\varepsilon \in \mathcal{O}(n)$ in the worst-case. The tree zipper comonad in Section 3.2.2.3 (p. 53) has $f^\dagger \in \mathcal{O}(n^2|f|)$.

{-# RULES "*extend/assoc*" $\forall\ g\ f\ .\ extend'\ (g \circ extend'\ f) = extend'\ g \circ extend'\ f$ #-}
{-# RULES "*codo*" $\forall\ h\ g\ f\ .\ extend'\ ((g \circ extend'\ f) \circ fmap'\ h) = extend'\ g \circ extend'\ f \circ fmap'\ h$ #-}

The first rule uses associativity of *extend* to eliminate nested *extend*s. The second rewrites *extend* of a desugared **codo**-block (of the form (40)), which is derived by naturality and associativity of *extend*. Currently, GHC's rewrite mechanism requires inlined aliases of class methods, *e.g.*:

{-# INLINE extend' #-}
$extend' :: Comonad\ c \Rightarrow (c\ a \to b) \to c\ a \to c\ b$
$extend' = extend$

Thus, the *codo* rule reduces the complexity from $\mathcal{O}(n^2|f|)$ to $\mathcal{O}(n|f|)$ assuming $(extend\ f) \in \mathcal{O}(n|f|)$. However, desugaring of each statement in a **codo**-block uses *fmap* inside of *extend*, which causes quadratic complexities still. More advanced rewriting to further improve the complexity of **codo** is further work.

4.3. Equational theory

The **codo**-notation provides *let*-binding, with no abstraction, and therefore resembles the simple L_1 language described in Section 2.2.2 (p. 27), which provided an internal language for a category with products. The equational theory for **codo** therefore resembles that of L_1 (p. 28), shown in **Figure** 4.2(a), following from the comonad laws. **Figure** 4.2(b) shows additional **codo** laws related to its desugaring.

Idempotency of the lax monoidal operation $czip :: (c\ a, c\ b) \to c\ (a, b)$ (discussed in Section 3.3) implies additional laws on **codo** relating tuple patterns and *czip* shown in **Figure** 4.2(c). Expressed in Haskell notation idempotency is:

$$czip\ (x, x) \equiv fmap\ (\lambda y \to (y, y))\ x. \tag{41}$$

For every rule involving a tuple pattern there is an equivalent rule derived using the (χ) rule (**Figure** 4.2(b)) which exchanges parameter and statement binders.

4.4. Comparing codo- and do-notation

While comonads and monads are dual, the **codo**- and **do**-notation do not appear dual. Both provide *let*-binding syntax, for composition of comonadic and monadic operations respectively. However, **codo**-blocks have a parameter, typed $c\ a \to b$ for a comonad c, whilst **do**-blocks are unparameterised, of type $m\ a$ for a monad m. Since comonads abstract functions with structured input the parameter to a **codo**-block is important. Monads abstract functions with structured output, thus the input is devoid of any additional structure, *i.e.*, *pure*. The pure input to monadic computations means that Haskell's scoping mechanism can be reused for handling the local variables in a **do**-block where inputs to the computation are then implicit; for the **codo**-notation the structured, comonadic context must be modelled manually since Haskell's scoping mechanism is pure.

(a) Comonad laws

[C1] $\mathbf{codo}\ x \Rightarrow f\ x$

$\equiv\ \mathbf{codo}\ x \Rightarrow y \leftarrow current\ x$
$f\ y$

[C2] $\mathbf{codo}\ x \Rightarrow f\ x$

$\equiv\ \mathbf{codo}\ x \Rightarrow y \leftarrow f\ x$
$current\ y$

[C3] (iff x is not free in e_1)
$\mathbf{codo}\ x \Rightarrow y \leftarrow e_1$
$z \leftarrow e_2$
e_3

$\equiv\ \mathbf{codo}\ x' \Rightarrow z \leftarrow (\mathbf{codo}\ x \Rightarrow y \leftarrow e_1$
$e_2)\ x'$
e_3

$\equiv\ \mathbf{codo}\ x' \Rightarrow y \leftarrow e_1$
$(\mathbf{codo}\ x \Rightarrow z \leftarrow e_2$
$e_3)\ x'$

(b) Pure laws

(η) $\mathbf{codo}\ x \Rightarrow f\ x \equiv f$

(β) $\mathbf{codo}\ x \Rightarrow z \leftarrow (\mathbf{codo}\ y \Rightarrow e_1)\ x$
e_2

$\equiv\ \mathbf{codo}\ x \Rightarrow y \leftarrow current\ x$
$z \leftarrow e_1$
e_2

(χ) $\mathbf{codo}\ p \Rightarrow e$

$\equiv\ \mathbf{codo}\ z \Rightarrow p \leftarrow current\ z$
e

(c) Additional laws – if eq. (41) holds

$\mathbf{codo}\ x \Rightarrow f\ a\ b$

$\equiv\ \mathbf{codo}\ x \Rightarrow (a', b') \leftarrow current\ (czip\ (a, b))$
$f\ a'\ b'$

$\mathbf{codo}\ (b, c) \Rightarrow f\ (czip\ (b, c))$

$\equiv\ \mathbf{codo}\ (b, c) \Rightarrow z \leftarrow (current\ b, current\ c)$
$f\ z$

Figure 4.2. Equational laws for the **codo**-notation

The **do**-notation corresponds roughly to a subset of an effectful language, providing a semantics for effectful *let*-binding embedded in Haskell. A monadic categorical semantics of *let*-binding in an effectful language is given by (*cf.* semantics in Section 2.3, p. 34):

$$\frac{[\![\Gamma \vdash e : \sigma]\!] = g : \Gamma \to \mathsf{M}\,\sigma \qquad [\![\Gamma, x : \sigma \vdash e' : \tau]\!] = g' : \Gamma \times \sigma \to \mathsf{M}\,\tau}{[\![\Gamma \vdash \mathbf{let}\ x = e\ \mathbf{in}\ e' : \tau]\!] : \Gamma \xrightarrow{\langle id, g\rangle} \Gamma \times \mathsf{M}\,\sigma \xrightarrow{\mathsf{st}} \mathsf{M}(\Gamma \times \sigma) \xrightarrow{bind\ g'} \mathsf{M}\,\tau} \tag{42}$$

In Haskell, all monads are *strong* with a canonical *strength* operator defined:

$$strength :: Monad\ m \Rightarrow (a, m\ b) \to m\ (a, b)$$
$$strength\ (a, mb) = mb \gg\!= (\lambda b \to return\ (a, b))$$

The definition of *strength* relies on Haskell binding and scoping features, where a is in scope for the function applied by $\gg\!=$. It is straightforwardly proved that this definition of *strength* satisfies the properties of a *strong monad* (see Definition B.1.9, p. 189 for these properties).

The standard translation of **do** can be derived from equation (42) by inlining the above definition of *strength* and simplifying according to the monad laws:

$$\frac{\Gamma \vdash e : m\ \sigma \qquad \Gamma, x : \sigma \vdash e' : m\ \tau}{\Gamma \vdash [\![\mathbf{do}\ x \leftarrow e; e']\!] \rightsquigarrow e \gg\!= (\lambda x \to e') : Monad\ m \Rightarrow m\ \tau}$$

giving a translation using just the monad operations and Haskell's scoping mechanism to define the semantics of multi-variable contexts for effectful *let*-binding. Thus the inputs to effectful computations are handled implicitly and so a **do**-block is just an expression of type $m\ a$.

For the **codo**-notation the situation is not so serendipitous. Recall the comonadic categorical semantics where $[\![\Gamma \vdash e : \tau]\!] : \mathsf{D}\Gamma \to \tau$. Thus, the context of free variables is structured where the notion of environments and scoping is inextricably linked with the comonad structure. Haskell's environments are *pure*, *i.e.*, not comonadic. Thus the **codo**-notation must model environments and scoping explicitly in its translation (Section 4.2) resulting in the relatively more complicated translation of **codo**-notation compared with that of **do**-notation.

4.5. Further examples

Numerical functions. Section 3.2.5 introduced the *exponent comonad*. The following shows a local operation of the exponent comonad, with domain $\mathbb{R}$, *i.e.*, for numerical functions, which defines a *local minima* predicate, by testing the first and second derivatives:

$$\begin{aligned} \mathit{minima} = \mathbf{codo}\ f \Rightarrow\ & f' \leftarrow \mathit{differentiate}\ f \\ & f'' \leftarrow \mathit{differentiate}\ f' \\ & (\mathit{current}\ f' \cong 0) \wedge (\mathit{current}\ f'' > 0) \end{aligned}$$

where $\cong$ tests approximate equality within some limit.

Labelled graphs. Many graph algorithms can be structured by a comonad, particularly compiler analyses and transformations on *control flow graphs* (*CFG*s). The following defines a labelled-graph comonad as a list of nodes which pair a label with a list of the connected vertices:

```
data LGraph a = LG [(a, [Int])] Int

instance Comonad LGraph where
  current (LG xs c) = fst (xs !! c)
  extend f (LG xs c) = LG (map (λc' → (f (LG xs c'), snd (xs !! c))) [0 .. length xs]) c
```

The *LGraph*-comonad resembles the array comonad where contexts are positions with a pointer-style cursor. Analyses over *CFGs* can be defined using graphs labelled by syntax trees. For example, a *live-variable* analysis can be defined, using the **codo**-notation, as:

```
lva = codo g ⇒ lv0 ← (defUse g, [])   -- compute definition/use sets, paired
               lva' lv0                -- with initial empty live-variable set

lva' = codo ((def, use), lv) ⇒
            live_out ← foldl union [] (successors lv)
            live_in  ← union (current def) ((current live_out) \\ (current use))
            lvp      ← ((current def, current use), current live_in)
            lvNext   ← lva' lvp
            if (lv ≡ live_in) then (current lv) else (current lvNext)
```

where *union* and set difference (\\) on lists have type $Eq\ a \Rightarrow [a] \to [a] \to [a]$ and *defUse* :: $LGraph\ AST \to ([Var], [Var])$ computes the sets of variables defined and used by each block in a CFG. The analysis is recursive, refining the set of live variables until a fixed-point is reached.

The live variables for every block of a *CFG* are computed by *extend lva*.

Lucid. Section 3.2.6 (p. 60) showed the *in context* comonad with **data** $InContext\ c\ a = InContext\ (c \to a)\ c$, where (**type** $Stream\ a = InContext\ Int\ a$) gives a succinct general stream comonad with a manifest cursor. This comonad is used here to embed first-order Lucid programs (without functions) in Haskell. Lucid's intensional operators can be defined as follows:

$$
\begin{aligned}
&fby :: Stream\ a \to Stream\ a \to a\\
&fby\ (InContext\ s\ c)\ (InContext\ t\ d) = \mathbf{if}\ (c \equiv 0 \wedge d \equiv 0)\ \mathbf{then}\ s\ 0\ \mathbf{else}\ t\ (d-1)\\
&next :: Stream\ a \to a\\
&next\ (InContext\ s\ c) = s\ (c+1)\\
&constant :: a \to Stream\ a\\
&constant\ x = InContext\ (\lambda\_ \to x)\ 0
\end{aligned}
$$

Uustalu and Vene previously provided a semantics for recursion in Lucid [**UV06**]. Here, Haskell's own recursive semantics can be reused. For example, the following uses **codo**-notation to define the stream of natural numbers (written in Lucid as $n = 0\ fby\ n + 1$):

$$
\begin{aligned}
&nat :: Stream\ a \to Int\\
&nat = \mathbf{codo}\ n \Rightarrow n' \leftarrow (nat\ n) + 1\\
&\qquad\qquad\qquad (constant\ 0)\ `fby`\ n'
\end{aligned}
$$

The stream of natural numbers can therefore be computed by $extend\ nat\ (constant\ ())$, where the constant stream of unit values provides the stream structure to *extend*:

$$extend\ nat\ (constant\ ()) \rightsquigarrow Stream\ \langle 0, 1, 2, \ldots \rangle\ 0$$

An alternate approach defines a fixed-point combinator. Since contextual computations are encoded as functions of type $\mathsf{D}\sigma \to \tau$ for some comonad D an appropriate fixed-point combinator has type $fix :: Comonad\ c \Rightarrow (c\ a \to a) \to c\ a$. A default definition is provided by $fix\ f = extend\ f\ (fix\ f)$, which can be used as follows:

$$
\begin{aligned}
&nat' = fix\ (\mathbf{codo}\ n \Rightarrow n' \leftarrow (extract\ n) + 1\\
&\qquad\qquad\qquad\qquad (constant\ 0)\ `fby`\ n')
\end{aligned}
$$

where $nat' :: Stream\ Int$. With Haskell's greatest-fixed point semantics, the default *fix* results in an undefined cursor where $nat' = Stream\ \langle 0, 1, 2, \ldots \rangle\ \bot$. An alternative definition for *fix* can be used to give the *least* fixed-point of the cursor, rather than the greatest ($\bot$) as follows:

$$fix\ f = \mathbf{let}\ (InContext\ x\ c) = extend\ f\ (fix\ f)\ \mathbf{in}\ InContext\ x\ 0$$

An example using *next* is the *howfar* operation (defined in the original Lucid book [**WA85**]) which takes an input stream of numbers and returns a stream of the "distance" to a value of 0

in the input stream at the corresponding position e.g.

$$howfar\ \langle 6,1,5,0,3,0,\ldots\rangle = \langle 3,2,1,0,1,0,\ldots\rangle$$

The *howfar* function can be written recursively as:

$$\begin{aligned} howfar = \mathbf{codo}\ x \Rightarrow x' \leftarrow next\ x \\ \qquad\qquad \mathbf{if}\ (current\ x \equiv 0)\ \mathbf{then}\ 0\ \mathbf{else}\ 1 + (howfar\ x') \end{aligned}$$

Alternate fixed-point operators are discussed briefly in Section 8.2.1 (p. 169).

4.6. Conclusion

Comonads provide a useful programming abstraction, essentially abstracting boilerplate code for traversing a data structure, allowing succinct definition of operations defined *locally* abstracting their promotion to *global* operations. The **codo**-notation presented here provides a lightweight syntax that considerably simplifies programming with comonads. Thus the **codo**-notation improves the *reading* and *writing* of contextual computations (see the *four Rs* in Chapter 1).

Another perspective on the notation is that it provides a contextual *sublanguage* embedded in Haskell, whose semantics is defined by a comonad. Since the **codo**-notation provides just contextual *let*-binding, the semantics requires no additional structure, *i.e.*, a monoidal structure is not necessary. However a lax monoidal functor is useful for passing multiple parameters to a **codo**-block as shown in Section 4.1.

Whilst monads are used widely in programming, comonads are relatively underused. This chapter postulated that a reason for this underuse is that, unlike monads, there is no language support to simplify programming with comonads. Thus, the use and experimentation of comonads as a design pattern in programming is impeded. The **codo**-notation therefore promotes the use of comonads in programming as a design pattern and tool for abstraction.

Chapter 7 uses the **codo**-notation in a language for contextual computations on containers.

Related work. The *arrow notation* in Haskell provides a syntax for programming with abstract notions of computation [**Pat01, Hug05**]. Syntax for application, abstraction, and binding is provided, abstracting operations for an encoding of categories in Haskell, with additional *arrow* operations for constructing computations and extending environments, defined by the *Category* and *Arrow* classes:

```
class Category cat where
  id :: cat x x
  (∘) :: cat y z → cat x y → cat x z

class Category a ⇒ Arrow a where
  arr :: (x → y) → a x y
  first :: a x y → a (x, z) (y, z)
```

Category thus captures composition and identity for its description of morphisms, encoded by the type *cat x y*. The *Arrow* class provides *arr* for promoting a Haskell function to the morphism type and *first* transforms a morphism to take and return an extra parameter, used for threading an environment through a computation. Other arrow combinators can be derived from this minimal set. CoKleisli categories can be encoded as follows:

$$\textbf{data}\ \mathit{CoKleisli}\ c\ x\ y = \mathit{CoK}\ \{\mathit{unCoK} :: (c\ x \to y)\}$$

$$\textbf{instance}\ \mathit{Comonad}\ c \Rightarrow \mathit{Category}\ (\mathit{CoKleisli}\ c)\ \textbf{where}$$
$$\quad \mathit{id} = \mathit{CoK}\ \mathit{current}$$
$$\quad (\mathit{CoK}\ g) \circ (\mathit{CoK}\ f) = \mathit{CoK}\ (g \circ (\mathit{extend}\ f))$$

$$\textbf{instance}\ \mathit{Comonad}\ c \Rightarrow \mathit{Arrow}\ (\mathit{CoKleisli}\ c)\ \textbf{where}$$
$$\quad \mathit{arr}\ k = \mathit{CoK}\ (k \circ \mathit{current})$$
$$\quad \mathit{first}\ (\mathit{CoK}\ f) = \mathit{CoK}\ (\lambda x \to (f\ (\mathit{cmap}\ \mathit{fst}\ x), \mathit{current}\ (\mathit{cmap}\ \mathit{snd}\ x)))$$

where *arr* pre-composes a function with *current*, and *first* is defined similarly to the handling of the local block environment in the desugaring of **codo**.

The arrow notation simplifies programming with arrows [**Hug05, Pat01**], comprising: *arrow formation*: $(\mathit{proc}\ x \to e)$, *arrow application*: $(f\ \texttt{-<}\ x)$ and *binding*: $(x \leftarrow e)$.

Using the coKleisli instances for *Category* and *Arrow*, comonadic operations can be written in arrow notation instead of **codo**. For example, the original *contours* example can be rewritten:

$$\begin{aligned}\mathit{contours} = \mathit{proc}\ x \to \textbf{do}\ & y \leftarrow \mathit{CoK}\ \mathit{gauss2D}\ \texttt{-<}\ x\\ & z \leftarrow \mathit{CoK}\ \mathit{gauss2D}\ \texttt{-<}\ y\\ & w \leftarrow \mathit{returnA}\ \texttt{-<}\ y - z\\ & \mathit{CoK}\ \mathit{laplace2D}\ \texttt{-<}\ w\end{aligned}$$

The arrow notation here is not much more complicated than **codo**, requiring just the additional overhead of the arrow application operator `-<` and lifting of *gauss2D* and *laplace2D* by *CoK*. Variables in the arrow notation here have non-comonadic type, *i.e.*, *Float* rather than $\mathit{CArray}\ (\mathit{Int}, \mathit{Int})\ \mathit{Float}$ for **codo** variables.

The arrow notation is however more cumbersome than **codo** when plumbing comonadic values, for example when using comonadic binary functions (of type $c\ t \to c\ t' \to t''$). The alternate definition of *contours* using *minus* becomes:

$$\begin{aligned}\mathit{contours}' = \mathit{proc}\ x \to \textbf{do}\ & y \leftarrow \mathit{CoK}\ \mathit{gauss2D}\ \texttt{-<}\ x\\ & z \leftarrow \mathit{CoK}\ \mathit{gauss2D}\ \texttt{-<}\ y\\ & w \leftarrow \mathit{CoK}\ (\lambda v \to \mathit{minus}\ (\mathit{fmap}\ \mathit{fst}\ v)\ (\mathit{fmap}\ \mathit{snd}\ v))\ \texttt{-<}\ (y, z)\\ & \mathit{CoK}\ \mathit{laplace2D}\ \texttt{-<}\ w\end{aligned}$$

where $v :: c\ (y, z)$ must be deconstructed manually. Whilst *minus* can be inlined here and the code rewritten to the more elegant first example, this is only possible since *minus* applies *extract* to both arguments. For other comonadic operations, with more complex behaviour, this refactoring is not always possible.

Comparing the two approaches, the *arrow* notation appears as powerful as the **codo**-notation, in terms of the computations that can be expressed. Indeed, from a categorical perspective, both notations need only a comonad structure (*i.e.*, coKleisli category) with no additional *closed* or *monoidal* structure (see Paterson's discussion [**Pat01**, §2.1]). However,

whilst the arrow notation is almost as simple as the **codo**-notation for some operations, the syntax is less natural for plumbing of comonadic values as seen above. In general, the **codo**-notation provides a more elegant, natural solution to programming with comonads in an applicative style (instead of the explicit application syntax of the arrow notation).

Further work. Whilst the **codo**-notation was developed here in Haskell, it could be applied in other languages. For example, a **codo**-notation for ML could be used to abstract laziness using the laziness comonad discussed in Section 3.2.5 (p. 58), with the type $C\ a = () \to a$.

As described in Section 4.2.1, further work is to improve the rewriting of **codo**-notation to provide further optimisations to the syntax.

Chapter 5

Generalising comonads

Whilst comonads capture a large class of contextual computations, as shown in Chapter 3, there are many reasonable notions of contextual computation that comonads do not abstract. This is mainly due to the shape preservation lemma (Section 3.4, p. 71) which shows that a context is uniform throughout a comonadic computation, manifested by the preservation of the *shape* of an incoming comonadic parameter (intension) throughout a computation. This contrasts with monads, where effects change the course of a computation manifested in the shape of the monadic results, which may change throughout a computation.

The primary contribution of this chapter is the *indexed comonad* structure which generalises comonads such that the shape preservation property is relaxed. Indexed comonads provide composition for functions of type $f : \mathsf{D}_R X \to Y$ which encode computations with contextual requirements (or demands) described by R. Thus D_R values provide the necessary context to satisfy the requirements R, where the shape(s) of D_R values may depend on R. Indexed comonads provide a composition:

$$(g : \mathsf{D}_R Y \to Z) \mathbin{\hat{\circ}} (f : \mathsf{D}_S X \to Y) : \mathsf{D}_{R \bullet S} X \to Z$$

where $\bullet$ is an operator on contextual requirements. Thus, the shape(s) of $\mathsf{D}_{R \bullet S}$ values may differ to those of D_R and D_S in the component computations f and g.

The next chapter uses indexed comonads (with additional structure) for the semantics of a contextual λ-calculus where computations are indexed by their requirements, called *coeffects*. A general *type and coeffect system* is introduced, dualising the traditional type and effect systems of Gifford and Lucassen [**GL86**], where the coeffects are computed by operations on indices.

Chapter structure and contributions. Indexed comonads are introduced in Section 5.4, along with examples of *partiality*, *product*, and *list* indexed comonads. Whilst indexed comonads are the primary contribution of this chapter, three other generalisations to comonads are introduced for capturing more general notions of contextual computation:

- *Relative comonads* – Monads and comonads are defined over endofunctors; they are monoids and comonoids in the category of endofunctors. The notion of a *relative monad* due to Altenkirch *et al.* generalises monads from endofunctors to functors [**ACU10, ACU11**]. Section 5.1 describes the dual of relative comonads and contributes an encoding of relative comonads in Haskell. This approach is particularly useful for categorical programming with comonad-like data types whose operations are not *parametrically* polymorphic but *ad-hoc* polymorphic. Relative comonads in Haskell are used later in Chapter 7.
- *Partial local operations* – In the mathematical definition, coKleisli morphisms are *total*. Some contextual computations may be defined only for particular contexts and are therefore partial. Partiality can be encoded by a monad which is integrated into the comonadic approach

by a *distributive law* between a monad and a comonad. This approach has had some use in the literature [**BS93, UV06**] and is used in Section 5.2 to encode contextual computations with specific *contextual requirements* as partial local operations. Composition then provides a notion of combining requirements, depending on the definition of the comonad and distributive law. This approach is a precursor to indexed comonads, which provides a related notion of composition that is both total and more flexible.

– *Lattices of comonads* – Section 5.3 describes *families* of comonads, where each comonad provides a particular contextual requirement. Given a semilattice structure on requirements, and therefore on comonads, a category is constructed, and thus a notion of composition, for morphisms with different contextual requirements, encoded by different comonads in the family. Composition is defined if ordered pairs of comonads have a *comonad morphism* between them. Indexed comonads in Section 5.4 generalise this category further, where Section 5.4.1 shows that lattices of comonads form an indexed comonad.

The *partial local operations* and *lattice of comonads* both relax the shape preservation property of comonads based on contextual requirements and are thus intermediate generalisations between comonads and indexed comonads. *Relative comonads* are a more independent generalisation.

5.1. Relative comonads

Haskell's default library provides two kinds of immutable array types: *boxed* and *unboxed*. Example 3.1.10 (p. 50) showed an array comonad in Haskell using the *boxed* array data type *Array*, where array elements are *non-strict*. Whilst array operations for boxed arrays are polymorphic in their element type, the operations of the *unboxed* array type *UArray* are restricted to primitive types (with fixed-size values) such as *Int*, *Bool*, and *Float*. This restriction is expressed and enforced with ad-hoc polymorphic operations on *UArray*, which are members of the *IArray* type class.

The *IArray* class captures the core operations of both boxed and unboxed arrays. *IArray* is parameterised by two types: an array type and an element type, *i.e.*, **class** *IArray a e* **where**.... Since the boxed array data type allows any element type its *IArray* instance is polymorphic in the element type, *i.e.*, **instance** *IArray Array e* **where**.... The unboxed array type on the other hand has several monomorphic instances *IArray UArray Int*, *IArray UArray Bool*, *etc.*.

This element restriction means that *UArray* cannot have an instance of *Comonad*, since the operations of *Comonad* are parametrically polymorphic in their element type (*i.e.*, unconstrained) whilst comonadic operations for *UArray* are constrained in their element type. The crux of the problem is that, whilst *Array* is analogous to an endofunctor, *UArray* is not endofunctor-like due to its element restrictions. Instead, *UArray* is analogous to a functor from a *subcategory* of **Hask** to **Hask**.[1]

[1]Recall that the notion of a category for Haskell types and functions **Hask** is informal.

5.1.1. Ad-hoc polymorphism and subcategories

A subcategory $\mathbb{S}$ of a category $\mathbb{C}$ comprises a subset (*subclass*) of the morphisms and objects of $\mathbb{C}$, where morphisms of $\mathbb{S}$ are strictly between $\mathbb{S}$ objects.

In the categorical programming approach, described in Section 2.4 (p. 38), parametrically polymorphic functions can be understood as models of families of morphisms indexed by objects. In Haskell, where type classes provide ad-hoc polymorphism, the instances of a type class describe a subset of types for which its methods are defined. Therefore, ad-hoc polymorphic functions are analogous to families of morphisms indexed by a *subcategory* of **Hask** objects.

For the *UArray* type, the *amap* function provides a morphism-mapping of a functor:

$$amap :: (Ix\ i, IArray\ UArray\ e, IArray\ UArray\ e') \Rightarrow (e \to e') \to UArray\ i\ e \to UArray\ i\ e'$$

where constraints $IArray\ UArray\ e, IArray\ UArray\ e'$ restrict the morphism mapping to morphisms of the *IArray UArray* subcategory which has only types/objects with an *IArray UArray* instance and morphisms $(IArray\ UArray\ a, IArray\ UArray\ b) \Rightarrow a \to b$. Therefore, *UArray* is not an endofunctor, and cannot be made an instance of *Functor* with $fmap = amap$, which causes a type error since *fmap* has no type class constraints on its elements. Since *UArray* is not an endofunctor it cannot have the additional structure of a comonad.

5.1.2. Comonads from (non-endo)functors

Although comonads (and monads) are defined over endofunctors (as comonoids/monoids in the category of endofunctors) their monoidal structure can be generalised to non-endofunctors. Altenkirch *et al.* defined such a structure for monads, called a *relative monad* [**ACU10**, **ACU11**]. The dual construction of a relative comonad is defined as follows:

Definition 5.1.1. A *relative comonad* over categories $\mathbb{K}$ and $\mathbb{C}$ comprises:

- a functor $\mathsf{K} : \mathbb{K} \to \mathbb{C}$
- an object mapping $\mathsf{D} : \mathbb{K}_0 \to \mathbb{C}_0$
- counit natural transformation: $\varepsilon_X : \mathsf{D}X \to \mathsf{K}X$
- coextension operation where for all $f : \mathsf{D}X \to \mathsf{K}Y \in \mathbb{C}_1$ then $f^\dagger : \mathsf{D}X \to \mathsf{D}Y \in \mathbb{C}_1$

with laws similar to those of the coKleisli laws (modulo the presence of K):

$$\text{[RcK1]}\quad (\varepsilon_A)^\dagger = id_{\mathsf{D}A} \qquad \text{[RcK2]}\quad \varepsilon_B \circ f^\dagger = f \qquad \text{[RcK3]}\quad (g \circ f^\dagger)^\dagger = g^\dagger \circ f^\dagger$$

Thus, relative comonads can be used to describe comonadic structures for data types that are restricted in their parametricity, taking $\mathbb{K}$ to be the subcategory of $\mathbb{C}$ which contains all those types that can be parameters of D. The functor K can be taken as the trivial *inclusion functor* which maps objects and morphisms from a subcategory back into its *super*category. Thus, K simply restricts the operations to element types X and Y in the subcategory $\mathbb{K}$.

(Non-endo)functors in Haskell. Subcategories, described by type-class constraints, can be defined as parameters to a structure by the use of *type-indexed families of constraints* (sometimes

called *constraint families* for brevity) in Haskell, which allow constraints on the types of a class method to vary per instance of the class (see Appendix D.4 for a more thorough introduction).

A class of functors (which may not be *endo*functors) can be defined in the following way, which was previously shown by Orchard and Schrijvers [**OS10**] and Bolingbroke [**Bol11**].

```
class RFunctor f where
  type SubCats f a :: Constraint
  type SubCats f a = ()
  rfmap :: (SubCats f a, SubCats f b) ⇒ (a → b) → f a → f b
```

The *SubCats* family allows a constraint to be specified per instance of the *RFunctor* class, with the empty (or *true*) constraint () as the default if none is specified. The constraints *SubCats f a* and *SubCats f b* therefore describe the source subcategory (and possibly a target subcategory if *f a* and *f b* are constrained).

Example 5.1.2. As discussed, the unboxed array type *UArray* is not analogous to an endofunctor, but is instead analogous to a functor from a subcategory of primitive types, described by instances of *IArray UArray*. *UArray* has the following instance of *RFunctor*:

```
instance RFunctor UArray where
  type SubCats UArray a = IArray UArray a
  rfmap = amap
```

Example 5.1.3. The *Set* data type in Haskell is implemented efficiently using balanced binary-trees, thus many of its operations require elements of a set to be *orderable* such that the internal tree-representation can be balanced [**Ada93**][2]. *Set* has a *map*-operation of type:

```
Set.map :: (Ord a, Ord b) ⇒ (a → b) → Set a → Set b
```

where the constraints (*Ord a*, *Ord b*) restrict the parametricity of *Set.map* to types with orderable values. Thus, *Set.map* describes a functor mapping from the *Ord*-subcategory of **Hask**.

```
instance RFunctor Set where
  type SubCats Set a = Ord a
  rfmap = Set.map
```

Relative comonads in Haskell. Similarly to the *RFunctor* class above, constraint families can be used to describe relative comonads in Haskell, as follows:

```
class RComonad d where
  type SubCats d x :: Constraint
  counit :: SubCats d x ⇒ d x → x
  coextend :: (SubCats d x, SubCats d y) ⇒ (d x → y) → d x → d y
```

[2] A non-efficient implementation of sets may still have type class constraints, e.g., *Eq* constraints so that repeated elements can be removed.

Similarly to *RFunctor*, *SubCats* can restrict the category $\mathbb{K}$ of the relative comonad to a subcategory of **Hask**. The functor K of the relative comonad is the inclusion functor $\mathsf{K} : \mathbb{K} \to \mathbf{Hask}$ which is implicit for x in *counit* and y in *coextend*, therefore K does not appear in the definition.

The *SubCats* constraint may also constrain $d\ x$ and $d\ y$, thus restricting the target category $\mathbb{C}$ also to a subcategory of **Hask** – the implications of this are not considered here.

Example 5.1.4. *Unboxed arrays* in Haskell can be defined as a relative comonad:

```
data UArr i a = UArr (UArray i a) i

instance Ix i ⇒ RComonad (UArr i) where
  type SubCats (UArr i) x = IArray UArray x
  counit (UArr arr c) = ...      -- same as current for the CArray comonad
  coextend f (UArr x c) = ...    -- same as extend for the CArray comonad
```

Example 5.1.5. The notion of a *pointed set*, common in topology, is a pair (S, x) of a set S with a distinguished element $x \in S$. Using Haskell's *Set* type, pointed sets have a relative comonad structure, defined:

```
data PSet a = PS (Set a) a
instance RComonad PSet where
  type SubCats PSet x = Ord x
  counit (PS s x) = x
  coextend f (PS s x) = PS (Set.map (λx' → f (PS (Set.delete x' s) x')) s) (f (PS s x))
```

5.1.3. Extending relative comonads in Haskell

The *SubCats* constraint for relative comonads in Haskell permits subcategory definitions of a particular style. The *RComonad* definition can be generalised further to allow more interesting subcategories as follows:

```
class RComonad d where
  type RObjs d x :: Constraint
  type RMorphs d x y :: Constraint
  counit :: RObjs d x ⇒ d x → x
  coextend :: (RMorphs d x y, RObjs d x, RObjs d y) ⇒ (d x → y) → d x → d y
```

where the specification of categories $\mathbb{K}$ and $\mathbb{C}$ is split between families *RObjs* and *RMorphs*. Since *counit* involves just objects of $\mathbb{K}$ and $\mathbb{C}$ the single parameter *RObjs* is used, specifying objects of the subcategories. However, for *coextend*, involving objects *and* morphisms of $\mathbb{K}$ and $\mathbb{C}$, *RMorphs* offers greater flexibility for specifying the morphisms of subcategories, allowing constraints relating the source and target types of morphisms. Since *RMorphs* is parameterised by x and y parameters, rather than $d\ x$ and y, then *RMorphs* can also describe restrictions

on a (relative) coKleisli category. For example, a comonad D can be defined where its coKleisli category is restricted to the subcategory of *endomorphisms*[3] (with the same source and target object) of **Hask** (using Haskell's *type equality constraint* [**CKJ05**]):

```
type RMorphs D x y = x∼y
```

Example 5.1.6 (Self-adjusting meshes)**.** Three-dimensional surfaces may be represented as two-dimensional arrays by *flattening* the surface, or *projecting* from points on the surface to a two-dimensional domain (*e.g.*, *stereographic projections* of spheres). This has the effect that elements of an array may represent sections of different surface area. For example, the surface of a torus can be represented by a two-dimensional array where indices are *wrapped* at both edges (see Section 7.1.4, p. 150). Elements on its top/bottom-edges cover the inner rim of the torus and are *smaller* on the conformed surface than the elements along the centre, covering the outer rim. Surfaces of this kind can be represented by the following data type:

```
data Surface a = Surface [((Int, Int), a)] (Int, Int) ((Int, Int) → a → a) ((Int, Int) → a → a)
```

The first and second components provide a usual array-like data type with data for the surface and a cursor. The last two components provide an isomorphism (projections) for computing the surface value at a particular position, and computing the projection value at that position.

This data type has the structure of a comonad whose coKleisli category is restricted to endomorphisms, since the projections have the parameter type a occurring in both covariant and contravariant positions. The comonad automatically adjusts values depending on the conformation of the surface in three-dimensions, defined in terms of *RComonad*:

```
instance RComonad Surface where
  type RObjs Surface x = ()
  type RMorphs Surface x y = x∼y
  counit (Surface dat c toSurface fromSurface) = toSurface c (unsafeLookup c dat)
  coextend k (Surface dat c toSurface fromSurface) =
    let k' (c', _) = (c', fromSurface c' (k (Surface dat c' toSurface fromSurface)))
    in Surface (map k' dat) c toSurface fromSurface
```

Thus, *counit* maps from a projection to a surface value. Indexing operations can be defined similarly which apply *toSurface* after indexing; *coextend* applies a local operation on the conformed surface, mapping the values back with *fromSurface* (or, the roles of *fromSurface* and *toSurface* may be swapped such that local operations are defined over the projection rather than surface).

This shows the flexibility of *RComonad* for specifying relative and regular comonads over subcategories of **Hask**. Chapter 7 uses similar comonads over subcategories of endomorphisms.

[3]Not to be confused with *endomorphism categories* [**BK00**, p.14], which tend to be defined as categories whose objects are pairs of objects and endomorphisms on those objects.

5.2. Partial computations and contextual requirements

In Chapter 3, coKleisli morphisms on **Set** are *total* functions, and are applied at all contexts by the extension operation of the comonad. However, a more general notion of context-dependent computation might have operations defined only for certain contexts, *i.e.*, having particular contextual requirements. These coKleisli morphisms are therefore *partial* which can be captured using the (*Maybe*/*option*) monad $\mathsf{M}A = A + 1$. For example, the following local operations on lists have non-exhaustive pattern matches, requiring lists of a minimum length:

$$
\begin{aligned}
&f\ (x : y : zs) = \mathit{Just}\ (\ldots) && \text{-- defined only for lists of at least length 2}\\
&f\ \_ \qquad\quad = \mathit{Nothing}\\
&g\ (x : y : z : w : vs) = \mathit{Just}\ (\ldots) && \text{-- defined only for lists of at least length 4}\\
&g\ \_ \qquad\qquad\ = \mathit{Nothing}
\end{aligned}
$$

Thus, f and g are both effectful *and* contextual computations, of the type $[a] \to \mathit{Maybe}\ b$. Such morphisms are those of a *biKleisli category* which provides composition for morphisms that are both monadic and comonadic, *i.e.*, of the form $\mathsf{D}A \to \mathsf{M}B$. A biKleisli category is defined by a Kleisli category over a coKleisli category, or equivalently a coKleisli category over a Kleisli category, with a distributive law between the monad and comonad, $\sigma : \mathsf{DM} \dot{\to} \mathsf{MD}$.

Definition 5.2.1. For a monad (M, η, μ) and a comonad $(\mathsf{D}, \varepsilon, \delta)$ a distributive law of D over M is the natural transformation $\sigma : \mathsf{DM} \dot{\to} \mathsf{MD}$ satisfying a number of axioms on the preservation of η, ε, μ, and δ by σ shown in the Appendix (Definition B.3.1, p. 191).

Previously, Brookes and Geva defined a categorical semantics using monads combined with comonads via a distributive law (using the terminology of a *double Kleisli*, rather than *biKleisli* category) capturing both intensional and effectful aspects of a program [**BS93**]. Harmer, Hyland, and Mellies used such a distributive law in context of defining strategies for game semantics [**HHM07**]. Uustalu and Vene use a biKleisli category for capturing stream computations which may have partial stream elements, where undefined elements are filtered out [**UV06**]. Their example resembles the following:

Example 5.2.2. The non-empty suffix list comonad distributes over the partiality monad with the following distributive law:

$$
\begin{aligned}
&\mathit{dist} :: [\mathit{Maybe}\ a] \to \mathit{Maybe}\ [a]\\
&\mathit{dist}\ (\mathit{Nothing} : \_) = \mathit{Nothing}\\
&\mathit{dist}\ xs \qquad\qquad\ = \mathit{Just}\ \$\ \mathit{filterNothing}\ xs
\end{aligned}
$$

where $\mathit{filterNothing} :: [\mathit{Maybe}\ a] \to [a]$ filters the *Nothing* values from a list, and projects values wrapped by *Just*, *e.g.*, $\mathit{dist}\ [\mathit{Just}\ 1, \mathit{Nothing}, \mathit{Just}\ 2] = \mathit{Just}\ [1, 2]$. Note that, if the head of the list (the cursor for the non-empty suffix list comonad) is *Nothing* then the whole result is *Nothing*. Proof of the axioms is straightforward and omitted here.

Definition 5.2.3. For a monad (M, η, μ) and comonad $(\mathsf{D}, \varepsilon, \delta)$ over the category $\mathbb{C}$, and a distributive law $\sigma : \mathsf{DM} \nrightarrow \mathsf{MD}$, then a *biKleisli* category ${}_{\mathsf{D}}\mathbb{C}_{\mathsf{M}}$ has objects $({}_{\mathsf{D}}\mathbb{C}_{\mathsf{M}})_0 = \mathbb{C}_0$ and morphisms ${}_{\mathsf{D}}\mathbb{C}_{\mathsf{M}}(A, B) = \mathbb{C}(\mathsf{D}A, \mathsf{M}A)$, with:

– composition: $(-\hat{\circ}-) : \mathbb{C}(\mathsf{D}Y, \mathsf{M}Z) \to \mathbb{C}(\mathsf{D}X, \mathsf{M}Y) \to \mathbb{C}(\mathsf{D}X, \mathsf{M}Z)$
$$g \mathbin{\hat{\circ}} f = (\mu \circ \mathsf{M}g) \circ \sigma \circ (\mathsf{D}f \circ \delta) \ (= g^* \circ \sigma \circ f^\dagger);$$
– identities: $\hat{id} : \mathbb{C}(\mathsf{D}A, \mathsf{M}A) = \eta \circ \varepsilon \quad (= \mathsf{M}\varepsilon \circ \eta\mathsf{D} = \varepsilon\mathsf{M} \circ \mathsf{D}\eta$ by [σ1,2], p. 191).

A biKleisli category can be seen as either an M-Kleisli category over a D-coKleisli category or equivalently as a D-coKleisli category over an M-Kleisli category. The proof is elided here but can be found in the work of Power and Watanabe [**PW02**, Corollary 6.6].

Therefore, a distributive law of the partiality monad over a comonad allows contextual computations defined only for particular contexts to be composed. The distributive law defines how those contexts that do not satisfy the contextual requirements affect the shape of the context throughout a computation, and thus defines how requirements are composed. Thus *shape preservation does not necessarily hold for the biKleisli category.*

For the example here of partial pattern matches on lists, contextual requirements are of the minimum number of elements in the context. The distributive law of Example 5.2.2 provides a biKleisli category where the following property on shape holds: given $f : [A] \to B$ which requires $\geqslant n$ elements more than the current element and $g : [B] \to C$ which requires $\geqslant m$ elements more than the current, then $g \mathbin{\hat{\circ}} f : [A] \to C$ requires $\geqslant m + n$ elements more than the current. Therefore for f defined on lists of a minimum length two and g on lists of a minimum length four, their composite $g \mathbin{\hat{\circ}} f$ is defined only for lists of a minimum length of five as shown by the following results on lists of length four and five respectively:

	$[a_0, a_1, a_2, a_3]$	$[a_0, a_1, a_2, a_3, a_4]$
δ	$[[a_0, a_1, a_2, a_3], [a_1, a_2, a_3], [a_2, a_3], [a_3]]$	$[[a_0, a_1, a_2, a_3, a_4], [a_1, a_2, a_3, a_4], [a_2, a_3, a_4], [a_3, a_4], [a_4]]$
$f^\dagger$	$[\mathit{Just}\, b_0, \mathit{Just}\, b_1, \mathit{Just}\, b_2, \mathit{Nothing}]$	$[\mathit{Just}\, b_0, \mathit{Just}\, b_1, \mathit{Just}\, b_2, \mathit{Just}\, b_3, \mathit{Nothing}]$
$\sigma \circ f^\dagger$	$\mathit{Just}\, [b_0, b_1, b_2]$	$\mathit{Just}\, [b_0, b_1, b_2, b_3]$
$g^* \circ \sigma \circ f^\dagger$	$\mathit{Nothing}$	$\mathit{Just}\, c_0$

Thus, the biKleisli category here allows the shape of the context to change.

For different notions of contextual requirements on lists the above distributive law and non-empty suffix comonad may not be appropriate. For example, consider contextual computations for a list comonad f and g, that are both defined only for lists of *exactly* length two. Their composition with the above comonad and distributive law is however always undefined since $f^\dagger\, [a_0, a_1] = [\mathit{Just}\, b_0, \mathit{Nothing}]$ and thus $\sigma \circ f^\dagger\, [a_0, a_1] = \mathit{Just}\, [b_0]$ and $g \circ \sigma \circ f^\dagger = \mathit{Nothing}$. A suitable comonad, where the composition would then be defined for lists of length two, would be the pointed-list comonad with an accompanying distributive law for the partiality monad.

This approach is useful, but it can be difficult to keep track of the contextual requirements since their composition is hidden within the comonad and distributive law. In Section 5.4, indexed comonads make contextual requirements explicit via their indices, where operations on indices describe the composition of requirements.

5.3. Lattices of comonads

Shape preservation is an unnecessarily restrictive property; there are situations where two contextual computations have different but related notions of context-dependence, and their exists a composition which combines the two contexts in some way.

Recall the product comonad $\mathsf{D}A = A \times X$, whose computations have an implicit parameter of type X, such as a configuration or a global resource. Shape preservation implies that the same implicit parameter is passed to every subcomputation. However, a different notion of composition might reasonably allow subcomputations to depend on different or additional parameters, where this dependency is propagated upwards through a computation such that the input provides the necessary parameters for all subcomputations. For example, consider the following two coKleisli morphisms of different product comonads, $(- \times (X \times Y))$ and $(- \times X)$:

$$\begin{aligned} f(a,(x,y)) &= \ldots \quad : (A \times (X \times Y)) \to B \\ g(a,x) &= \ldots \quad\quad\; : (B \times X) \to C \end{aligned}$$

The following composite of g and f is possible since $f^\dagger$ provides the X value needed by g:

$$A \times (X \times Y) \xrightarrow{f^\dagger} B \times (X \times Y) \xrightarrow{id \times \pi_1} B \times X \xrightarrow{g} C$$

Thus, f and g are composed so that their composite requires the union of their contexts. This, composition is not supported by the standard comonad structure.

Similarly, given two coKleisli morphisms, *e.g.*, $p : (A \times X) \to B$ and $q : (B \times (X \times Y)) \to C$, *i.e.*, where p has less parameters and q has more parameters, their composite can be defined:

$$A \times (X \times Y) \xrightarrow{(p \circ (id \times \pi_1))^\dagger} B \times (X \times Y) \xrightarrow{q} C$$

where the composite has the union of two contexts for p and q as its input.

In both examples $(id \times \pi_1) : (A \times (X \times Y)) \to (A \times X)$ provides a *comonad morphism* from the comonad with more information to the comonad with less. Comonad morphisms are defined:

Definition 5.3.1. A *comonad (homo)morphism* between comonads $(\mathsf{D}, \varepsilon, \delta)$ and $(\mathsf{D}', \varepsilon', \delta')$ is a natural transformation $\iota : \mathsf{D} \dot{\to} \mathsf{D}'$ such that:

$$\begin{array}{ccccccc} A & \xleftarrow{\varepsilon_A} & \mathsf{D}A & \xrightarrow{\delta_A} & \mathsf{D}\mathsf{D}A & \xrightarrow{\iota_{\mathsf{D}A}} & \mathsf{D}'\mathsf{D}A \\ & \nwarrow^{\varepsilon'_A} & \downarrow{\scriptstyle \iota_A} & & & & \uparrow{\scriptstyle \mathsf{D}'\iota_A} \\ & & \mathsf{D}'A & \xrightarrow[\delta'_A]{} & \mathsf{D}'\mathsf{D}'A & & \end{array} \tag{43}$$

In the above example compositions, a single comonad morphism was used. Two coKleisli morphisms defined on unrelated product comonads, $f : A \times X \to B$ and $g : B \times Y \to C$, where X and Y share no common component, can be composed using two comonad morphisms:

$$A \times (X \times Y) \xrightarrow{(f \circ (id \times \pi_1))^\dagger} B \times (X \times Y) \xrightarrow{(id \times \pi_2)} B \times X \xrightarrow{g} C$$

with comonad morphisms $(id \times \pi_1)$ and $(id \times \pi_2)$. The first two examples are a special case of this example, where one of the comonad morphisms was the identity morphism.

Since the product comonad distributes over itself in a CCC (Section 3.2.7 (p. 60)), an alternative approach composes two product comonads, *e.g.*, $(- \times X)$ and $(- \times Y)$ to $((- \times X) \times Y)$, via a distributive law. This would still require the above comonad morphisms to *lift* coKleisli morphisms to the composite comonad, thus the distributive law does not provide any advantages.

A semilattice of product comonads. The product comonad can be extended to a family of product comonads, indexed by the implicit parameter type X with $\mathsf{D}_X = - \times X$. Any two product comonads in this family can be related by a common least-upper bound $\sqcup$, where in the first two examples $(X \times Y) \sqcup X = X \times Y = Y \sqcup (X \times Y)$, and in the second, $X \sqcup Y = X \times Y$ (if X and Y are distinct ground objects, *i.e.*, they do not decompose further). The unit type 1 is the identity of the least-upper bound (*i.e.*, the least element). Given some notion of "syntactic ordering" for objects in $\mathbb{C}$ then $\sqcup$ can be defined such that is associative, commutative, and idempotent (where syntactic ordering means that, for example, $X \sqcup Y = Y \sqcup X = X \times Y$, where the ordering fixes which object is the first component). A full definition of $\sqcup$ is not given here for brevity. The family of product comonads therefore has a *bounded join-semilattice* structure.

Definition 5.3.2. A *join semilattice* $(S, \sqcup)$ comprises a set S and a total binary operation $\sqcup : S \times S \to S$ (least-upper bound) such that for all $x, y, z \in S$ the following conditions hold:

$$\begin{array}{ll} (\textit{associativity}) & x \sqcup (y \sqcup z) = (x \sqcup y) \sqcup z \\ (\textit{commutativity}) & x \sqcup y = y \sqcup x \\ (\textit{idempotency}) & x \sqcup x = x \end{array}$$

A *bounded join-semilattice* $(S, \sqcup, \bot)$ has a *least* element $\bot \in S$ such that for all $x \in S$:

$$(\textit{identity}) \quad x \sqcup \bot = x$$

Remark 5.3.3. Every join semilattice $(S, \sqcup)$ induces a partial order $(\sqsubseteq) \subseteq S \times S$ where:

$$x \sqsubseteq y \;\; \textit{iff} \; x \sqcup y = y \tag{44}$$

For the product comonad, when $X \sqsubseteq Y$ there is a comonad morphism $\iota^Y_X : (A \times Y) \to (A \times X)$, essentially defining a projection from a comonad with more information to one with less.

Category of a semi-lattice of comonads. The semilattice of product comonads forms a category with a general notion of composition provided by the comonad morphisms. This can be generalised to any semilattice of comonads.

Definition 5.3.4. An *indexed family of endofunctors* is a functor $\mathsf{F} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$ from an index category $\mathbb{I}$ to the category of endofunctors on $\mathbb{C}$. Thus, for every object $X \in \mathbb{I}_0$, $\mathsf{F}X : \mathbb{C} \to \mathbb{C}$ is an endofunctor, and for every morphism $f : X \to Y \in \mathbb{I}_1$ then $\mathsf{F}f : \mathsf{F}X \dot{\to} \mathsf{F}Y$ is a natural transformation. From now on, where F is an indexed family of endofunctors, the endofunctor $\mathsf{F}X$ is written as F_X instead.

The term *indexed family of (endo)functors* might be abbreviated to *indexed functors* although a different notion of indexed functor is found in topos theory, with functors between *indexed categories* (which are themselves functors); see for example, the work of Johnstone [**Joh02**].

Definition 5.3.5. An *indexed family of comonads* comprises an indexed family of endofunctors $\mathsf{D} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$ where for every $X \in \mathbb{I}_0$ there is a comonad $(\mathsf{D}_X, \delta_X, \varepsilon_X)$.

Indexed families of comonads are different to indexed comonads introduced later in Section 5.4.

Definition 5.3.6. A *bounded semilattice of comonads* $(\mathsf{D}, \sqcup, \bot)$ comprises an indexed family of comonads $\mathsf{D} : \mathbb{J}^{\mathsf{op}} \to [\mathbb{C}, \mathbb{C}]$ and a semilattice structure $(\mathbb{J}_0, \sqcup, \bot)$. The category $\mathbb{J}$ is taken as the partial order $(\mathbb{J}_0, \sqsubseteq)$, thus $X \sqsubseteq Y$ implies the unique morphism $f : Y \to X \in \mathbb{J}_1^{\mathsf{op}}$ (see Example A.1.5 (p. 181) for partial orders as categories), for which there is a comonad morphism $\mathsf{D}_f = \iota_X^Y : \mathsf{D}_Y \dot{\to} \mathsf{D}_X$.

Whilst indexed families of endofunctors are defined in terms of $\mathsf{F} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$, a bounded semilattice of comonads has $\mathbb{I} = \mathbb{J}^{\mathsf{op}}$ such that $\mathsf{D}_f : \mathsf{D}_Y \dot{\to} \mathsf{D}_X$ is defined as the comonad morphism ι_X^Y. The implication is that the partial order $X \sqsubseteq Y$ on $\mathbb{J}_0$ implies that D_X contains less information than D_Y, thus a well behaved comonad morphism must map a comonad with more information to one with less: $\mathsf{D}_f : \mathsf{D}_Y \dot{\to} \mathsf{D}_X$.

Definition 5.3.7. Given a bounded semilattice of comonads $(\mathsf{D}, \sqcup, \bot)$, there is a category ${}_{\mathsf{D}}\mathbb{C}$ with objects ${}_{\mathsf{D}}\mathbb{C}_0 = \mathbb{C}_0$ and morphisms ${}_{\mathsf{D}}\mathbb{C}(A, B) = \bigcup_{X \in \mathbb{I}_0} \mathbb{C}(\mathsf{D}_X A, B)$ with identities $id_A : \mathsf{D}_\bot A \to A = (\varepsilon_\bot)_A$ and composition defined:

$$\hat{\circ} : (\mathsf{D}_Y B \to C) \to (\mathsf{D}_X A \to B) \to (\mathsf{D}_{(X \sqcup Y)} A \to C)$$
$$g \mathbin{\hat{\circ}} f = g \circ \iota_Y^{X \sqcup Y} \circ (f \circ \iota_X^{X \sqcup Y})^\dagger \tag{45}$$

where the extension $(-)^\dagger$ in the above composition (45) is that of the comonad $\mathsf{D}_{(X \sqcup Y)}$, *i.e.*:

$$\mathsf{D}_{(X \sqcup Y)} A \xrightarrow{\delta_{X \sqcup Y}} \mathsf{D}_{(X \sqcup Y)} \mathsf{D}_{(X \sqcup Y)} A \xrightarrow{\mathsf{D}_{(X \sqcup Y)} \iota_X^{X \sqcup Y}} \mathsf{D}_{(X \sqcup Y)} \mathsf{D}_X A \xrightarrow{\mathsf{D}_{(X \sqcup Y)} f} \mathsf{D}_{(X \sqcup Y)} B \xrightarrow{\iota_Y^{X \sqcup Y}} \mathsf{D}_Y B \xrightarrow{g} C$$

Morphisms of two different comonads are therefore composed by precomposing the morphisms with comonad morphisms from the least-upper bound comonad. Appendix C.3.1 (p. 202) provides the proof of the laws of a category for this construction.

Example 5.3.8. Product comonads form a semilattice of comonads $\mathsf{D}_X = (- \times X)$ with the previously described semilattice structure $(\mathbb{C}_0, \sqcup, 1)$, providing a category where $f : (A \times X) \to B, g : (B \times Y) \to C$ compose to $g \mathbin{\hat{\circ}} f : (A \times (X \times Y)) \to C$.

Example 5.3.9. Length-indexed non-empty lists with safe pointers (Section 3.2.3, p. 55), $\mathsf{List} : \mathbb{N}_{>0} \to [\mathbb{C}, \mathbb{C}]$, provide an indexed family of comonads, with $\mathsf{List}_N A = A^N \times N$, *e.g.* $\mathsf{List}_1 A = A \times \{0\}$ and $\mathsf{List}_2 A = A \times A \times \{0, 1\}$. Members of this family have comonad morphisms such as the following $\iota_2^3 : \mathsf{List}_3 \dot{\to} \mathsf{List}_2$ (using the notation $[...]@c$ for a list paired with a cursor c):

$$\iota_2^3\ [x, y, z]@0 = [x, y]@0$$
$$\iota_2^3\ [x, y, z]@1 = [y, z]@0$$
$$\iota_2^3\ [x, y, z]@2 = [y, z]@1$$

The coherence conditions of such comonad morphisms are easily proved.

This family of comonads has a semilattice structure $(\mathbb{N}_{>0}, \mathsf{max}, 0)$ (with the $\leqslant$ ordering), *i.e.*, the least-upper bound of two list comonads is the maximum of the two. Thus, morphisms $f : \mathsf{List}_M A \to B$ and $g : \mathsf{List}_N B \to C$ are composed to $g \mathbin{\hat{\circ}} f : \mathsf{List}_{(M \,\mathsf{max}\, N)} A \to C$.

Remark 5.3.10. For a particular index X, a subcategory of a category for a lattice of comonads that has only morphisms $\mathbb{C}(\mathsf{D}_X A, B)$ collapses to the usual coKleisli category for the (shape preserving) comonad D_X under the idempotence of $\sqcup$ (*i.e.*, $X \sqcup X = X$) and the property that $\iota^X_X = id_{\mathsf{D}_X A}$ is the identity comonad morphism.

The next section introduces the further generalisation of *indexed comonads*, subsuming semilattices of comonads.

5.4. Indexed comonads

Indexed types are often used in programming to describe not just *what* values a program computes, but also *how* they are computed. For the lattice of product comonads, comonads are indexed by their implicit parameter type revealing exactly what values are required from the environment; the standard product comonad enforces the same parameter on all subcomputations, whereas the product comonad lattice allows computations to, for example, require no implicit parameters with the unit index 1. Morphisms for the category of the product comonad lattice thus describe more accurately how their values are computed. Similarly, for the length-indexed list comonad family, the length-index specifies the minimum number of list values required.

The category from a (semi)lattice of comonads can be generalised, where members of the family of endofunctors $\mathsf{D} : \mathbb{I}^{\mathsf{op}} \to [\mathbb{C}, \mathbb{C}]$ are *not* required to be comonads. The indexed comonad structure then arises from a *monoidal functor* structure over D.

Definition 5.4.1. An *indexed comonad* comprises an indexed family of endofunctors $\mathsf{D} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$, where $\mathbb{I}$ is a *strict monoidal category*[4] $(\mathbb{I}, \bullet, I)$ (Definition (2.2.6), p. 29), and where D is a *colax monoidal functor* with natural transformations n_1 and $\mathsf{n}_{A,B}$ (Definition 3.3.1, p. 63), which provide the operations ε_I and $\delta_{X,Y}$:

- $\mathsf{n}_1 = \varepsilon_I : \mathsf{D}_I \mathrel{\dot{\to}} 1_{\mathbb{C}}$
- $\mathsf{n}_{A,B} = \delta_{X,Y} : \mathsf{D}_{X \bullet Y} \mathrel{\dot{\to}} \mathsf{D}_X \, \mathsf{D}_Y$

The colax monoidal functor D between monoidal categories $([\mathbb{C}, \mathbb{C}], \circ, 1_{\mathbb{C}})$ and $(\mathbb{I}, \bullet, I)$ has the following coherence conditions (dualising the diagrams for lax monoidal functors, Appendix 74):

$$\begin{array}{ccc}
\mathsf{D}_{X \bullet (Y \bullet Z)} & \xleftarrow{\mathsf{D}\alpha_{X,Y,Z}} & \mathsf{D}_{(X \bullet Y) \bullet Z} \\
{\scriptstyle \delta_{X, Y \bullet Z}}\downarrow & & \downarrow{\scriptstyle \delta_{X \bullet Y, Z}} \\
\mathsf{D}_X \circ \mathsf{D}_{Y \bullet Z} & & \mathsf{D}_{X \bullet Y} \circ \mathsf{D}_Z \\
{\scriptstyle (\mathsf{D}_X) \circ \delta_{Y,Z}}\downarrow & & \downarrow{\scriptstyle \delta_{X,Y} \circ (\mathsf{D}_Z)} \\
\mathsf{D}_X \circ (\mathsf{D}_Y \circ \mathsf{D}_Z) & \xleftarrow{\alpha_{\mathsf{D}_X, \mathsf{D}_Y, \mathsf{D}_Z}} & (\mathsf{D}_X \circ \mathsf{D}_Y) \circ \mathsf{D}_Z
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{D}_{X \bullet I} & \xrightarrow{\mathsf{D}\rho} & \mathsf{D}_X \\
{\scriptstyle \delta_{X,I}}\downarrow & & \uparrow{\scriptstyle \rho} \\
\mathsf{D}_X \circ \mathsf{D}_I & \xrightarrow[\mathsf{D}_X \circ \varepsilon_I]{} & \mathsf{D}_X \circ 1_{\mathbb{C}}
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{D}_{I \bullet X} & \xrightarrow{\mathsf{D}\lambda} & \mathsf{D}_X \\
{\scriptstyle \delta_{I,X}}\downarrow & & \uparrow{\scriptstyle \lambda} \\
\mathsf{D}_I \circ \mathsf{D}_X & \xrightarrow[\varepsilon_I \circ \mathsf{D}_X]{} & 1_{\mathbb{C}} \circ \mathsf{D}_X
\end{array}
\tag{46}$$

[4] Recall that for a strict monoidal category, the associativity and unit operations α, λ, ρ are identities, rather than natural transformations.

where α, λ, and ρ are natural isomorphisms for associativity and unitality in each monoidal category. Since both $([\mathbb{C},\mathbb{C}], \circ, 1_{\mathbb{C}})$ and $(\mathbb{I}, \bullet, I)$ are strict monoidal categories[5] then α, λ, ρ (for each category) are identities, thus the colax monoidal functor conditions (46) simplify to:

$$\begin{array}{ccc} \mathsf{D}_X & \xrightarrow{\delta_{X,I}} & \mathsf{D}_X\mathsf{D}_I \\ {\scriptstyle \delta_{I,X}}\downarrow & \overset{[C1]}{\searrow} \; \overset{[C2]}{\searrow} & \downarrow{\scriptstyle \mathsf{D}_X\varepsilon_I} \\ \mathsf{D}_I\mathsf{D}_X & \xrightarrow[\varepsilon_I\mathsf{D}_X]{} & \mathsf{D}_X \end{array} \qquad \begin{array}{ccc} \mathsf{D}_{X\bullet Y\bullet Z} & \xrightarrow{\delta_{X\bullet Y,Z}} & \mathsf{D}_{X\bullet Y}\mathsf{D}_Z \\ {\scriptstyle \delta_{X,Y\bullet Z}}\downarrow & [C3] & \downarrow{\scriptstyle \delta_{X,Y}\mathsf{D}_Z} \\ \mathsf{D}_X\mathsf{D}_{Y\bullet Z} & \xrightarrow[\mathsf{D}_X\delta_{Y,Z}]{} & \mathsf{D}_X\mathsf{D}_Y\mathsf{D}_Z \end{array} \tag{47}$$

which are analogous to the standard conditions of a comonad for coassociativity and counits, but subject to preservation of the monoidal structure of indices.

Example 5.4.2. The *indexed partiality comonad* can be used to encode contextual computation that may or may not use their context, comprising:

- index category $\mathbb{B}$, which is *discrete* (only identity morphisms) with $\mathbb{B}_0 = \{\mathsf{T}, \mathsf{F}\}$ *i.e.* booleans;
- strict monoidal structure $(\mathbb{B}, \wedge, \mathsf{T})$, *i.e.*, logical conjunction;
- indexed family of endofunctors $\mathsf{P} : \mathbb{B} \to [\mathbb{C},\mathbb{C}]$ defined:
  - $\mathsf{P}_\mathsf{T} A = A$ and $\mathsf{P}_\mathsf{T} f = f$ (identity functor)
  - $\mathsf{P}_\mathsf{F} A = 1$ and $\mathsf{P}_\mathsf{F} f = id_1$ (constant functor)
- operations $\delta_{X,Y}$ and ε_T defined:

$$\begin{array}{llll} \delta_{\mathsf{F},\mathsf{F}} & : \mathsf{P}_{\mathsf{F}\wedge\mathsf{F}} \dot{\to} \mathsf{P}_\mathsf{F}\mathsf{P}_\mathsf{F} & = id_1 & \qquad \delta_{\mathsf{T},\mathsf{F}} \; : \mathsf{P}_{\mathsf{T}\wedge\mathsf{F}} \dot{\to} \mathsf{P}_\mathsf{F}\mathsf{P}_\mathsf{T} \; = !_A \\ \delta_{\mathsf{F},\mathsf{T}} & : \mathsf{P}_{\mathsf{F}\wedge\mathsf{T}} \dot{\to} \mathsf{P}_\mathsf{T}\mathsf{P}_\mathsf{F} & = id_1 & \qquad \delta_{\mathsf{T},\mathsf{T}} \; : \mathsf{P}_{\mathsf{T}\wedge\mathsf{T}} \dot{\to} \mathsf{P}_\mathsf{T}\mathsf{P}_\mathsf{T} \; = id_A \\ \varepsilon_\mathsf{T} & : \mathsf{P}_\mathsf{T} \dot{\to} 1 & = id_A & \end{array}$$

The axioms of indexed comonads for these operations is easily proved, *e.g.*, for [C1,2] from D_X:

- $X = \mathsf{T}$, then $\varepsilon_I\mathsf{P}_\mathsf{T} \circ \delta_{I,\mathsf{T}} = \mathsf{P}_\mathsf{T}\varepsilon_I \circ \delta_{\mathsf{T},I} = id_A$
- $X = \mathsf{F}$, then $\varepsilon_I\mathsf{P}_\mathsf{F} \circ \delta_{I,\mathsf{F}} = \mathsf{P}_\mathsf{F}\varepsilon_I \circ \delta_{\mathsf{F},I} = !_A$

The indexed partiality comonad abstracts composition for computations that may or may not use the context (*i.e.*, partiality in the input), where those that do not are more precisely modelled as a constant morphism $\mathsf{P}_\mathsf{F} A \to B$. The disjoint union of the members P corresponds to the standard partiality type $\mathsf{P}A = A + 1$ (the *Maybe* type in Haskell and *option* type in ML).

Indexed coKleisli categories & triples. Analogous notions of coKleisli extension and categories can be defined for indexed comonads.

Definition 5.4.3. Given an indexed comonad $\mathsf{D} : \mathbb{I} \to [\mathbb{C},\mathbb{C}]$ its *indexed coKleisli category* ${}_\mathsf{D}\mathbb{C}$ has objects ${}_\mathsf{D}\mathbb{C}_0 = \mathbb{C}_0$ and morphisms ${}_\mathsf{D}\mathbb{C}(X,Y) = \bigcup_{R\in\mathbb{I}_0} \mathbb{C}(\mathsf{D}_R X, Y)$ with:

- *composition* - $g \mathbin{\hat{\circ}} f = g \circ \mathsf{D}_R f \circ \delta_{R,S}$ (for all $g : \mathsf{D}_R Y \to Z, f : \mathsf{D}_S X \to Y \in \mathbb{C}_1$)
- *identities* - $\hat{id}_A = \varepsilon_{I\,A}$

Analogously to comonads and coKleisli triples, indexed comonad have an equivalent presentation as *indexed coKleisli triples*.

[5] $[\mathbb{C},\mathbb{C}]$ is a monoidal category where $\otimes = \circ$, *i.e.* the tensor operation is functor composition, and $I = 1_\mathbb{C}$, *i.e.* the identity functor on $\mathbb{C}$, and which is strict since $\mathsf{F}1_\mathbb{C} = \mathsf{F} = 1_\mathbb{C}\mathsf{F}$ and $\mathsf{F}(\mathsf{G}\mathsf{H}) = (\mathsf{F}\mathsf{G})\mathsf{H}$.

Definition 5.4.4. An *indexed coKleisli triple* comprises an indexed family of object mappings providing the operations ε_I and $(-)^\dagger_{R,S}$:

- *counit*: $\varepsilon_{I\,A} : \mathsf{D}_I A \to A$
- *coextension*: for all $f : \mathsf{D}_S A \to B \in \mathbb{C}_1$ then $f^\dagger_{R,S\,A,B} : \mathsf{D}_{R \bullet S} A \to \mathsf{D}_R B \in \mathbb{C}_1$

satisfying analogous axioms to the coKleisli triples.

Example 5.4.5. The *indexed partiality coKleisli triple* has ε_{T} and $(-)^\dagger_{S,T}$ defined:

$$\begin{array}{llllll} f^\dagger_{\mathsf{F},\mathsf{F}} & = id_1 & f^\dagger_{\mathsf{T},\mathsf{F}} & = f \circ !_A & \varepsilon_{\mathsf{T}} & = id_A \\ f^\dagger_{\mathsf{F},\mathsf{T}} & = id_1 & f^\dagger_{\mathsf{T},\mathsf{T}} & = f & & \end{array}$$

Analogous to the equivalence between coKleisli triples and comonads in comonoidal form, indexed coKleisli triples are equivalent to indexed comonads, with the equalities:

$$f^\dagger_{R,S} = \mathsf{D}_R f \circ \delta_{R,S} \qquad \delta_{R,S} = (id_{\mathsf{D}_S})^\dagger_{R,S} \qquad \mathsf{D}_R f = \mathsf{D}_R(f \circ \varepsilon_{\mathsf{T}}) \circ \delta_{R,\mathsf{T}} \tag{48}$$

Example 5.4.6. In Girard *et al.*'s bounded linear logic (BLL) the *bounded reuse* of a proposition A is written $!_X A$, meaning it may be reused at most X times [**GSS92**]. A model for $!_X A$ is suggested as the X-times tensor of A, *e.g.*, $!_X A = A \times \ldots \times A$ (or, $!_X A = A^X$). The *storage axiom* of BLL corresponds roughly to coextension of an indexed comonad for the ! modality:

$$\text{Storage}\frac{!_{\bar{y}}\Gamma \vdash A}{!_{x\bar{y}}\Gamma \vdash !_x A}$$

Note that $\bar{y}$ is a vector with $\bar{y} = y_1, \ldots, y_n$ for $|\Gamma| = n$ (this is discussed later in Section 6.6.2, p. 140). Bounded reuse is intuitively modelled by an indexed comonad $! : \mathbb{N} \to [\mathbb{C}, \mathbb{C}]$, with $!_N A = A^N$, the natural-number multiplication monoid $(\mathbb{N}, \times, 1)$, and:

$$\begin{aligned} f^\dagger_{X,Y}\langle a_1, .., a_{XY}\rangle &= \langle f\langle a_1, .., a_Y\rangle, f\langle a_{Y+1}, .., a_{Y+Y}\rangle, .., f\langle a_{(X-1)Y+1}, .., a_{XY}\rangle\rangle \\ \varepsilon_1\langle a_1\rangle &= a_1 \end{aligned}$$

Let $f : \mathsf{D}_Y A \to B$, which uses its parameter Y times. Its extension, where the result B is used X times, $f^\dagger_{X,Y} : \mathsf{D}_{X \times Y} A \to \mathsf{D}_X B$, requires Y copies of the parameter A to compute each B, therefore $X \times Y$ copies of the parameter are needed; counit $(\varepsilon_1)_A : !_1 A \to A$ uses its parameter once. This indexed comonad captures notions of data access and usage of the context.

Relating comonads and indexed comonads. Indexed comonads collapse to regular comonads when $\mathbb{I}$ is a single-object monoidal category (*i.e.*, if $\mathbb{I}_0 = \{*\}$ then D_* is a comonad). Thus, all comonads are indexed comonads with a trivial index structure.

However, not all indexed comonads are comonads, such as the indexed partiality comonad. Further, for all $X \in \mathbb{I}_0$, D_X is not necessarily a comonad (*e.g.*, $D_I A = 1$ in Example 5.4.2). Whilst comonads exhibit shape preservation, where the shape of intermediate contexts in a comonadic computation are uniform, indexed comonads may not be shape preserving; shape preservation is relaxed. For example, for all $R, S, f : \mathsf{D}_S A \to B$ then $\mathsf{D}_R !_B \circ f^\dagger_{R,S} \neq \mathsf{D}_{R \bullet S} !_A$, since the parameter to D may determine shape (recall that $\mathsf{D}_R !_A : \mathsf{D}_R A \to \mathsf{D}_R 1$ computes the shape). *Shape* may thus change *monoidally* as described by the $(\mathbb{I}, \bullet, I)$ structure.

5.4.1. Semilattices of comonads as indexed comonads

Section 5.3 showed a category derived from families of comonads with a bounded semilattice structure. From this structure an indexed comonad can be constructed.

Proposition 5.4.7. *Given an indexed family of comonads* $\mathsf{D} : \mathbb{I}^{\mathsf{op}} \to [\mathbb{C},\mathbb{C}]$ *with bounded join semilattice structure* $(\mathbb{I}, \sqcup, \bot)$ *(inducing the partial order* $\sqsubseteq$*), then for every* $f : R \to S \in \mathbb{I}_1$ *there is a comonad morphism* $\iota^S_R = \mathsf{D}_f : \mathsf{D}_S \to \mathsf{D}_R$*. Therefore* $(\mathbb{I}^{\mathsf{op}}, \sqcup, \bot)$ *is a strict monoidal category and* D *is an indexed comonad with operations:*

$$\begin{aligned} \hat{\varepsilon}_\bot &:= \mathsf{D}_\bot \xrightarrow{\varepsilon_\bot} 1_{\mathbb{C}} \\ \hat{\delta}_{S,T} &:= \mathsf{D}_{S\sqcup T} \xrightarrow{\delta_{S\sqcup T}} \mathsf{D}_{S\sqcup T}\mathsf{D}_{S\sqcup T} \xrightarrow{\mathsf{D}_{S\sqcup T}\iota^{S\sqcup T}_T} \mathsf{D}_{S\sqcup T}\mathsf{D}_T \xrightarrow{\iota^{S\sqcup T}_S \mathsf{D}_T} \mathsf{D}_S\mathsf{D}_T \end{aligned} \tag{49}$$

For the monoidal category with bifunctor $\sqcup$, the signature of $\sqcup$ on pairs of morphisms $f : A \to B, g : X \to Y \in \mathbb{I}_1^{\mathsf{op}}$ implies that if $A \sqsupseteq B$ and $X \sqsupseteq Y$ then $(A \sqcup X) \sqsupseteq (B \sqcup Y)$ for all $A, B, X, Y \in \mathbb{I}_0$ (which follows from the join semi-lattice laws). The associativity, unit, and symmetric operations $\alpha, \lambda, \rho, \gamma$ of the monoidal category are identities following from the bounded join-semilattice properties. Thus, $\hat{\delta}_{S,T}$ and $\hat{\varepsilon}_\bot$ define a colax monoidal functor providing the operations of an indexed comonad where the coherence conditions (47) follow from the category construction in Definition 5.3.7 (and the proof of the category axioms in Appendix C.3.1, p. 202).

Example 5.4.8. The *indexed product comonad* is the canonical indexed comonad on $\mathsf{D} : \mathbf{Set}^{\mathsf{op}} \to [\mathbb{C},\mathbb{C}]$, with $\mathsf{D}_X = (- \times X)$, following from Proposition 5.4.7 and a lattice of comonads $(- \times X)$ following from the bounded join-semilattice $(\mathbb{C}, \cup, \emptyset)$.

Example 5.4.9. The *indexed pointed-list comonad* is the canonical indexed comonad $\mathsf{List}_N = A^N \times N$ following from Proposition 5.4.7 and the bounded join-semilattice of comonads List_N (Example 5.3.9 (p. 104)) with the bounded join-semilattice $(\mathbb{N}_{>0}, \mathsf{max}, 0)$.

Remark 5.4.10. The *indexed partiality comonad* in Example 5.4.2 is not an example of an indexed comonad induced from a family of comonads since P is not a family of comonads as $\mathsf{P}_{\mathsf{F}}A = 1$ does not have a comonad structure.

5.5. Conclusion

5.5.1. Related work

In the literature there are various notions of indexed or parameterised monads. For comparison, indexed comonads can be dualised in the obvious way to an indexed monad.

Definition 5.5.1. Stated briefly, for a base category $\mathbb{C}$ and an index category $\mathbb{I}$ which is a strict monoidal category $(\mathbb{I}, \bullet, I)$, an *indexed monad* comprises an indexed family of functors $\mathsf{M} : \mathbb{I} \to [\mathbb{C},\mathbb{C}]$ where M is a *lax* monoidal functor (*cf.*, colax for indexed comonads) providing the natural transformations: $\eta_I : 1_{\mathbb{C}} \dot{\to} \mathsf{M}_I$ and $\mu_{S,T} : \mathsf{M}_S\,\mathsf{M}_T \dot{\to} \mathsf{M}_{S \bullet T}$, satisfying the dual of the coherence conditions for indexed comonads (equation (46), p. 104).

As an informal example, the usual notion of *state monad* can be delimited with sets of the *read* and *write* effects where M can be refined with respect to the effect, *e.g.*, $\mathsf{M}\{\mathsf{read}\,\rho : \tau\}\,A = \tau \to A$ and $\mathsf{M}\{\mathsf{write}\,\rho : \tau\}\,A = A \times \tau$ (note: the latter is not itself a monad), or for mixed reading/writing $\mathsf{M}_X A = X \to (A \times X)$.

Previously, Wadler and Thiemann gave a syntactic correspondence between type-and-effect systems and a monadic semantics by annotating monadic type constructors with effect sets, M^F [**WT03**]. They gave soundness results between the effect system and operational semantics, and conjectured a coherent denotational semantics of effects and monads. One suggestion was to associate to each effect set F a different monad M^F. The indexed monad structure here provides a semantic correspondence between type-and-effect systems and a monadic semantics, but where each M^F may not be a monad. In the next chapter, the notion of type-and-coeffect system are introduced, to which indexed comonads provide the semantics.

Indexed categories, functors, and monads. The notion of an indexed category and indexed functor appears in topos theory and elsewhere [**Joh02**]. This notion of indexed functor differs to the one here, where an indexed endofunctor, written $\mathsf{M}^X : \mathcal{C}^X \to \mathcal{C}^X$, maps between indexed categories $\mathcal{C}^X$ (themselves defined as functors $\mathcal{C} : \mathbb{I}^{\mathsf{op}} \to \mathbf{Cat}$).

Indexed families of functors in this chapter, on some category $\mathbb{C}$, may be described by indexed functors $\mathsf{M}^X : \mathcal{C}^X \to \mathcal{C}^X$ with constant indexed categories $\mathcal{C}A = \mathbb{C}$ for the category $\mathbb{C}$. However, given such an indexed endofunctor, *indexed monads* are usually defined as a triple $(T^X, \eta^X : 1_{\mathbb{C}} \dot{\to} T^X, \mu : T^X T^X \dot{\to} T^X)$, thus the index is constant over the monad structure [**CR91**]. This differs to the indexing notions used here where an indexed (co)monad arises from a (co)lax monoidal functor, where indices are not constant but have a monoidal structure preserved by the operations of the (co)monad.

Parameterised notions of computation. Atkey describes parameterised strong monads, which have a functor $\mathsf{M} : \mathbb{S}^{\mathsf{op}} \times \mathbb{S} \times \mathbb{C} \to \mathbb{C}$ where $\mathbb{S}$ is a category of *state descriptions* as objects and *state manipulations* as morphisms [**Atk09**]. An object $\mathsf{M}(S_1, S_2, A)$ captures a computation that starts in state S_1 and ends in state S_2, yielding a value A. By contravariance (*i.e.* from $\mathbb{S}^{\mathsf{op}}$), the pre-condition S_1 can be strengthened, and by covariance the post-condition S_2 can be weakened. The operations of a parameterised monad are then:

$$\begin{aligned} \textit{(unit)} \quad & \eta_A^S : A \to \mathsf{M}(S, S, A) \\ \textit{(multiplication)} \quad & \mu_A^{S_1,S_2,S_3} : \mathsf{M}(S_1, S_2, (\mathsf{M}(S_2, S_3, A))) \to \mathsf{M}(S_1, S_3, A) \\ \textit{(strength)} \quad & \tau_{A,B}^{S_1,S_2} : A \times \mathsf{M}(S_1, S_2, B) \to \mathsf{M}(S_1, S_2, A \times B) \end{aligned}$$

where the state descriptor parameters are written here as superscripts. Thus, multiplication requires that the inner and outer computations agree on a mutual intermediate state S_2. An example parameterised strong monad models typed state with the monad $\mathsf{M}(S_1, S_2, A) = (S_2 \times A)^{S_1}$, *i.e.*, the state monad where the parameterisations to the monad indicate the type of the incoming and outgoing state [**Atk09**][pp.6-7]. McBride uses Atkey's parameterised monads to parameterise computations with specifications in Hoare type theory [**McB11**].

Atkey makes the observation that, whilst monads are monoids (in $[\mathbb{C},\mathbb{C}]$), parameterised monads are *partial* monoids (*i.e.*, *categories*) where multiplication (composition) is defined only for some objects that "agree" in some sense. As an example, Atkey generalises the product monad ($\mathsf{M}A = X \times A$ where X is a monoid) to parameterised product monads with a (small) category $\mathbb{S}$ as the parameter, where $\mathsf{M}(S_1, S_2, A) = \mathbb{S}(S_1, S_2) \times A$ (again $\mathsf{M} : \mathbb{S}^{\mathsf{op}} \times \mathbb{S} \times \mathbb{C} \to \mathbb{C}$). The unit and multiplication are defined $\eta^S_A(a) = (id_s, a)$ and $\mu^{S_1,S_2,S_3}_A(g,(f,a)) = (g \circ f, a)$. Thus, multiplication is defined only when the functions g and f compose in $\mathbb{S}$.

Since categories are partial monoids, parameterised monads could be embedded into an indexed monad for $((\mathcal{S}\times\mathcal{S}^{\mathsf{op}})+\{\bot, I\}, \bullet', I)$ where $(S1, S2)\bullet'(S2', S3) = (S1, S3)$ when $S2 = S2'$ and $\bot$ otherwise, and where I is a distinguished element representing a universal identity.

Embedding indexed monads into parameterised monads seems more difficult. Monoids can be encoded as single-object categories with elements of the monoid as morphisms, but μ is parameterised by object tuples, not morphisms. Further work is to explore unifying the two approaches, generalising monads and comonads to be indexing by monoids and categories.

5.5.2. Concluding remarks

From the perspective of reasoning about program behaviour, both monads and comonads *over-approximate* impurity in a computation, providing a black-box view of effects and context-dependence. For comonads, a function of type $f : \mathsf{D}A \to B$ reveals it has some notion of context-dependence, encoded by D, but no more information about the exact contextual requirements of f. For example, if $\mathsf{D} = \mathsf{Array}$, f might access elements in neighbouring contexts, *e.g.*, for a one-dimensional array with cursor i, f might access $i-1$ and $i+1$, or perhaps only the current element i in which case the computation is trivially context dependent.

This overapproximation provided by both monads and comonads contrasts with the *type and effect systems* of Gifford, Jouvelot, Lucassen and Talpin [**GL86, TJ92**] which give a precise account of the effects incurred by a computation. This section generalised comonads in a number of ways, most notably to *indexed comonads* which, similarly to effect systems, make precise the context-dependence (contextual requirements) of a computation. The next chapter considers a dual notion to effect systems, called *coeffect* systems, which precisely describe the contextual requirements of a computation. Indexed comonads, with some additional structure, are used to give the semantics of the *coeffect calculus* where the indices of a computation are its coeffect.

Relative indexed comonads. Relative comonads and indexed comonads shown in this chapter can be unified into *indexed relative comonads*, with a family of functors $\mathsf{D} : \mathbb{I} \to [\mathbb{K}, \mathbb{C}]$, a functor $\mathsf{K} : \mathbb{K} \to \mathbb{C}$ and analogous operations to the usual indexed comonad operations. Note that K is not indexed. Chapter 7 uses relative indexed comonads.

The **codo**-notation of Chapter 4 can be generalised easily to relative comonads, replacing the usual *extend* operation with the relative comonad extensions operation shown above. Chapter 7 uses a version of the **codo**-notation for relative (indexed) comonads.

Chapter 6

The coeffect calculus

Chapter 3 introduced the contextual λ-calculus and its semantics in terms of monoidal comonads. It was shown that the shape preservation property of comonads limits this calculus to computations where the context has the same shape throughout. Various other notions of context (see Chapter 1) require a more flexible contextual language and semantics. To this end, the previous chapter introduced *indexed comonads* which generalise comonads and provide composition for computations with potentially different, but related, contextual requirements; indexed comonads do not exhibit shape preservation.

This chapter introduces a calculus of contextual computations called the *coeffect calculus* with a semantics given by indexed comonads with additional structure. This calculus provides a more precise account of context than the *contextual λ-calculus* of Chapter 3, where the denotations of its expressions capture exactly the contextual requirements. The propagation of contextual requirements through a computation is tracked and described for this calculus by a novel class of static analyses called *coeffect systems*, dualising effect systems of Gifford and Lucassen [**GL86**]. Coeffect systems encode language invariants, aid correctness, and provide information for optimisation. The next chapter introduces a language for efficient and correct stencil computations which is an instance of the coeffect calculus.

Coeffect systems are joint work with Tomáš Petříček and Alan Mycroft. Some of the material from this chapter appears in a condensed form in the paper *Coeffects: Unified Static Analysis of Context-Dependence* published in the proceedings of ICALP 2013 [**POM13**].

Introduction. Consider a language whose programs can access *resources* provided by the execution context. The semantics of the language can be given in terms of the indexed product comonad, introduced in the previous chapter, with some additional structure, described later in this chapter. The resources accessed by an expression are a contextual requirement, the propagation of which can be tracked by a *coeffect system*, which reconstructs contextual properties of a program. Coeffect systems dualise effect systems.

Effect systems, introduced by Gifford and Lucassen to track memory effects [**GL86**], are traditionally described as syntax-directed analyses by typing rules augmented with effect judgements, *i.e.*, $\Gamma \vdash e : \tau \,!\, F$ where F describes the effects of e – usually a set of effect tokens. Coeffect systems similarly follow the inference structure of typing rules, with coeffects written on the left of the entailment $\vdash$, *i.e.*, associated with the context, of form: $\Gamma \,?\, F \vdash e : \tau$.

Resource access in the example language has the following syntax and coeffect/type rule:

$$[\textsc{access}] \quad \Gamma \,?\, \{\mathsf{A}\} \vdash \mathsf{access}\,\mathsf{A} : \tau$$

Coeffects of a compound expression, *e.g.*, addition, are the union of the subexpression's coeffects:

$$[+]\frac{\Gamma \,?\, F_1 \vdash e_1 : \mathbb{R} \qquad \Gamma \,?\, F_2 \vdash e_2 : \mathbb{R}}{\Gamma \,?\, F_1 \cup F_2 \vdash e_1 + e_2 : \mathbb{R}}$$

The key difference between coeffect and effect systems is the propagation of (co)effect information for λ-abstraction. Traditionally, λ-abstraction is pure with respect to side-effects. Effect systems therefore annotate a λ-term with no information and encapsulate the effects of the function body as *latent* effects which occur only when the function is applied, with the rule:

$$[\text{ABS}]\frac{\Gamma, v : \sigma \vdash e : \tau \,!\, F}{\Gamma \vdash \lambda v.e : \sigma \xrightarrow{F} \tau \,!\, \emptyset}$$

where $\xrightarrow{F}$ is the function type constructor for functions with latent effects F.

For coeffects, the rule for λ-abstraction is not dual: the coeffects of a function body are *split* between the definition context and latent coeffect:

$$[\text{ABS}]\frac{\Gamma, v : \sigma \,?\, F \sqcap F' \vdash e : \tau}{\Gamma \,?\, F \vdash \lambda v.e : \sigma \xrightarrow{F'} \tau}$$

The latent coeffect describes contextual requirements that are exposed when the function is applied. Reading [ABS] bottom-up, contextual requirements of the function body are thus fulfilled by a combination (greatest-lower bound) of the definition context and application context. Reading the rule top-down, the contextual requirements of a function body are split between the defining context and application context.

This example resource language could be extended to a distributed computing environment where the [ABS] rule then allows functions to be defined that require resources that are not available in the defining context but are instead available in a different context, such as another device to which the function is sent. This example will discussed further in Section 6.2.3.

Chapter structure and contributions. Section 6.1 begins with an overview of the literature on effect systems and the use of *annotated monads* to unify effect systems and monadic type systems. This chapter then makes the following contributions:

- Section 6.2 introduces a number of example contextual languages with coeffect systems which describe the semantics of context. The examples are: a language which optimises *dead code* where the coeffect system is essentially a *liveness* analysis, a (causal) dataflow language which is optimised to cache the minimum number of elements required from the input stream, and a language of rebindable resources similar to the example above.
- The general *coeffect calculus* is introduced with a general coeffect system parameterised by a *coeffect algebra* structure (Section 6.3). The examples of Section 6.2 are shown as specialisation of this general calculus.
- Section 6.4 defines a categorical semantics for the coeffect calculus in terms of an indexed comonad with some additional structure where Section 6.4.3 shows the conditions on the additional category-theoretic structure to provide equational theories of β- and η-equivalence.
- Section 6.5 shows that *implicit parameters* feature of Haskell can be described in the coeffect calculus. Furthermore, this section shows how coeffect systems that have set-based contextual requirements can be encoded using Haskell's type class constraint mechanism. This technique is used in the next chapter to encode a coeffect system in Haskell.

$$[\textsc{var}]\frac{v : \tau \in \Gamma}{\Gamma \vdash v : \tau \,!\, \bot} \quad [\textsc{const}]\ \Gamma \vdash c : \tau \,!\, \bot \quad [\textsc{let}]\frac{\Gamma \vdash e_1 : \tau_1 \,!\, F \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \,!\, F'}{\Gamma \vdash \mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 : \tau' \,!\, F \sqcup F'}$$

$$[\textsc{abs}]\frac{\Gamma, v : \sigma \vdash e : \tau \,!\, F}{\Gamma \vdash \lambda v.e : \sigma \xrightarrow{F} \tau \,!\, \bot} \quad [\textsc{app}]\frac{\Gamma \vdash e_1 : \sigma \xrightarrow{F''} \tau \,!\, F \qquad \Gamma \vdash e_2 : \sigma \,!\, F'}{\Gamma \vdash e_1\, e_2 : \tau \,!\, F \sqcup F' \sqcup F''}$$

(a) Core rules

$$[\textsc{write}]\frac{\Gamma \vdash e_1 : \texttt{ref}\ \rho\ \tau \,!\, F \qquad \Gamma \vdash e_2 : \tau \,!\, F'}{\Gamma \vdash \texttt{set}\ e_1\, e_2 : () \,!\, F \cup F' \cup \{(\mathsf{write}, \rho)\})} \quad [\textsc{read}]\frac{\Gamma \vdash e : \texttt{ref}\ \rho\ \tau \,!\, F}{\Gamma \vdash \texttt{get}\ e : \tau \,!\, F \cup \{(\mathsf{read}, \rho)\}}$$

(b) Effect-specific rules for memory access

Figure 6.1. Gifford-Lucassen effect system for an impure λ-calculus, in modern notation.

6.1. Background

Effect systems classify expressions by their type, *i.e.*, what result is computed, and their effect, *i.e.*, how the result is computed. Gifford and Lucassen's initial motivation for effect systems was to integrate both pure, higher-order functional programming and imperative programming in one "*fluent*" language [**GL86, LG88**]. Effect systems have since been used to analyse many different notions of effect, such as memory access [**GL86**], atomicity in concurrency [**FQ03**], communication in message-passing systems [**JG89a**], and *control side-effects* from unstructured control primitives (*goto* and *comefrom*) encoded by continuations [**JG89b**].

The traditional presentation of effect systems can be divided into *core* and *application-specific* rules. Core rules apply generally to any notion of effect, describing effect propagation through a λ-calculus. Application-specific rules describe the effects of language-specific syntax. Gifford and Lucassen described a lattice structure on effects, where the join (least-upper bound) operation combines effect annotations and the least-element annotates pure expressions [**GL86**]. Many systems are however simply lattices of sets. **Figure** 6.1(a) shows the core rules with *let*-binding, generalised from sets of effects to an arbitrary bounded join-semilattice $(\mathcal{P}(F), \sqcup, \bot)$ (see Definition 5.3.2 (p. 102)). Originally, effects were considered for the polymorphic λ-calculus with type and effect polymorphism. For simplicity, the presentation here is monomorphic, although the rules using variables can be instantiated at different types and effects.

The [VAR] and [CONST] rules annotate variables and constants as pure, with effect annotation $\bot$. Likewise, the [ABS] rule explains that constructing a function has no side-effects, where the side effects of the function body are encapsulated as *latent* effects. The [APP] rule explains that the effects of application are a combination of the effects of the parameter, function, and the function's latent effects once applied. The [LET] rule explains that the semantics of **let** evaluates both the bound and receiving terms, thus the effect information of each is combined. **Figure** 6.1(b) shows the effect-specific rules for memory operations, using the particular join-semilattice $(\mathcal{P}(\{\mathsf{write}, \mathsf{read}\} \times \mathcal{R}), \cup, \emptyset)$ where $\mathcal{R}$ is a set of regions.

Later systems introduced *subeffecting* allowing effects to be overapproximated with respect to some effect ordering [**TJ92**]:

$$[\text{SUB}]\frac{\Gamma \vdash e : \tau \,!\, F \qquad F \subseteq F'}{\Gamma \vdash e : \tau \,!\, F'}$$

Effect systems and equational theories. As mentioned in Section 2.3.2, effectful languages do not have the same β- and η-equalities as the pure λ-calculus. The unsoundness of (call-by-name) β-equality is exposed by the rules of the general effect system. For example, assume [$\equiv$-β]:

$$[\equiv\text{-}\beta]\frac{\Gamma, x : \sigma \vdash e : \tau \,!\, F \qquad \Gamma \vdash e' : \sigma \,!\, F'}{\Gamma \vdash (\lambda x.e)\, e' \equiv e[x := e'] : \tau \,!\, F \sqcup \mathsf{F}'}$$

This rule implies a meta-rule for substitution with effects. This however leads to a contradiction where, by the definition of syntactic substitution $(\lambda v.e)[x := e'] \equiv \lambda v.e[x := e']$, then:

$$\frac{\dfrac{\Gamma, x : \tau', v : \sigma \vdash e : \tau \,!\, F \quad \Gamma \vdash e' : \tau' \,!\, F'}{\Gamma, v : \sigma \vdash e[x := e'] : \tau \,!\, F \sqcup F'}}{\Gamma \vdash \lambda v.e[x := e'] : \sigma \xrightarrow{F \sqcup F'} \tau \,!\, \bot} \quad \neq \quad \frac{\dfrac{\Gamma, x : \tau', v : \sigma \vdash e : \tau \,!\, F}{\Gamma, x : \tau' \vdash (\lambda v.e) : \sigma \xrightarrow{F} \tau \,!\, \bot} \quad \Gamma \vdash e' : \tau' \,!\, F'}{\Gamma \vdash (\lambda v.e)[x := e'] : \sigma \xrightarrow{F} \tau \,!\, F'}$$

thus violating β-equality. Therefore if β-reduction is included in the operational semantics then type preservation (subject reduction) cannot hold. Instead, a restricted substitution is provided for substitution of pure (syntactic) values, *i.e.*,

Lemma 6.1.1. *If* $\Gamma, x : \tau' \vdash e : \tau \,!\, F$ *and* $\Gamma \vdash e' : \tau' \,!\, \bot$, *and* $\mathit{value}(e')$ *then* $\Gamma \vdash e[x := e'] : \tau \,!\, F$.

The lack of η-equality for effectful languages is evident from the following effect judgment:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash e : \sigma \xrightarrow{F} \tau \,!\, F'}{\Gamma, x : \sigma \vdash e : \sigma \xrightarrow{F} \tau \,!\, F'}\,[\text{WEAKEN}] \qquad \dfrac{x : \sigma \in (\Gamma, x : \sigma)}{\Gamma, x : \sigma \vdash x : \sigma \,!\, \bot}\,[\text{VAR}]}{\Gamma, x : \sigma \vdash e\ x : \tau \,!\, F \sqcup F'}\,[\text{APP}]}{\Gamma \vdash \lambda x.e\ x : \sigma \xrightarrow{F \sqcup F'} \tau \,!\, \bot}\,[\text{ABS}]$$

where the effects of the premise expression become latent effects, thus η-equality does not hold since the premise and conclusion here should match.

Monads and effects. Recall that (strong) monads abstract the composition of effects, encoded in a pure language by a functor M (equivalently, data type) (Section 2.3.1, p. 35). Apart from the semantic content of monads, monads also give a coarse-grained account of whether an expression has an effect or not, *i.e.*, an expression with a monadic type is *possibly* effectful, and an expression without a monadic type is *definitely* pure. Comparatively, effect systems provide a fine-grained account of effects. Wadler and Thiemann unified the approaches of monads and effect systems by annotating monadic types with effect information [**WT03**] (or in their words "*transposing effects to monads*") with judgments of the form $\Gamma \vdash e : \mathsf{M}^F \tau$ for some strong monad M which is *internal* to the type theory, *i.e.*, M appears as a type constructor. Their unification is sound since the propagation of effect information in an effect system corresponds exactly to the semantic propagation of effects in a strong monadic semantics; for example, abstraction is pure as described by effect systems and witnessed by the monadic semantics.

$$[\text{VAR}]\frac{}{\Gamma, v : \tau \vdash v : \mathsf{M}^{\bot}\tau} \qquad [\text{LET}]\frac{\Gamma, v : \sigma \vdash e_2 : \mathsf{M}^{F}\tau \quad \Gamma \vdash e_1 : \mathsf{M}^{F'}\sigma}{\Gamma \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \mathsf{M}^{F \sqcup F'}\tau}$$

$$[\text{ABS}]\frac{\Gamma, v : \tau \vdash e : \mathsf{M}^{F}\tau'}{\Gamma \vdash \lambda v.e : \mathsf{M}^{\bot}(\tau \to \mathsf{M}^{F}\tau')} \qquad [\text{APP}]\frac{\Gamma \vdash e_1 : \mathsf{M}^{F}(\tau \to \mathsf{M}^{F''}\tau') \quad \Gamma \vdash e_2 : \mathsf{M}^{F'}\tau}{\Gamma \vdash e_1\ e_2 : \mathsf{M}^{F \sqcup F' \sqcup F''}\tau'}$$

$$[\text{SUB}]\frac{\Gamma \vdash e : \mathsf{M}^{F}\tau \quad F \sqsubseteq F'}{\Gamma \vdash e : \mathsf{M}^{F'}\tau} \qquad [\text{IF}]\frac{\Gamma \vdash e : \mathsf{M}^{F}\mathsf{bool} \quad \Gamma \vdash e_1 : \mathsf{M}^{F'}\tau \quad \Gamma \vdash e_2 : \mathsf{M}^{F''}\tau}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \mathsf{M}^{F \sqcup F' \sqcup F''}\tau}$$

Figure 6.2. Effect system transposed to annotations on monadic types

Figure 6.2 shows the core rules of effect systems from **Figure** 6.1(a) transposed to monadic annotations, and includes a rule for conditionals that overapproximates the effects since the branch-taken may not be statically known. The general presentation of **Figure** 6.1(a) using a bounded semi-lattice is used, rather than the Wadler and Thiemann's presentation using sets composed by $\cup$ with $\emptyset$ for the pure effect.

Richer effect algebras. Nielson and Nielson defined effect systems using richer algebraic structures than mere sets with union or semilattices. They separated the traditional approach of a lattice of effects into operations for sequential composition, alternation, and fixed-points [**NN99**]. This captures a wider range of analyses, including *may* and *must* properties. This chapter takes a similar approach, defining a flexible system with an algebraic structure on coeffects.

6.2. Example contextual calculi with coeffect systems

The following introduces three contextual languages, all based on the simply-typed λ-calculus, each with an accompanying coeffect system. Section 6.2.1 describes a language with no additional syntax and a coeffect analysis of *live variables*. Section 6.2.2 describes a simple causal dataflow language (in the style of Lucid) with a coeffect analysis of data access on the context (whose semantics is provided by streams). Section 6.2.3 describes the example language of the introduction with resources (such as a database, GPS, *etc.*) from the execution context and a coeffect analysis of resource requirements.

As described, coeffect judgments are written $\Gamma \,?\, F \vdash e : \tau$ with latent coeffect annotations on function types $\xrightarrow{F}$. Each language allows *sub-coeffecting* with the following rule for some $\sqsubseteq$:

$$[\text{SUB}]\frac{\Gamma \,?\, R \vdash e : \tau \quad R \sqsubseteq R'}{\Gamma \,?\, R' \vdash e : \tau}$$

Additionally, every system admits structural rules which preserve coeffects. A *structural* contextual calculus, where structural rules are not admissible, is considered briefly as further work in Section 6.6.2 (p. 140), which does not have this property.

Each of these languages has a categorical semantics in terms of some indexed comonad with additional structure, where $[\![\Gamma \,?\, R \vdash e : \tau]\!] : \mathsf{D}_R[\![\Gamma]\!] \to [\![\tau]\!]$. Discussion of the categorical semantics is delayed till Section 6.4.

The rules of a coeffect system can be read in two ways. Reading rules *bottom-up* gives an intuition of how the semantic notion of context (encoded by a functor D_T for coeffect T) is propagated to subterms. Reading the rules *top-down* gives an intuition of how contextual requirements are composed and propagated upwards towards the root of an expression.

6.2.1. Free-variables and liveness

The first example is of the simply-typed λ-calculus, with constants, *let*-binding, and conditionals, with no notion of context other than the free-variables. However, its semantics eliminates dead-code based on its coeffect system which calculates a live-variable analysis.

The standard live-variable (or *liveness*) analysis calculates which free variables in an expression may be used and which are definitely not used. Expressions bound to variables which are definitely not used are therefore *dead*, *i.e.*, do not contribute any information to the result (in a pure language), and can thus be eliminated [**App97**]. Liveness is a coeffect since it is a property of the free-variable context of an expression. Wadler describes this property of a variables that is *ignored* as the *absence* of a variable [**Wad88**].

Often this analysis is presented for imperative languages defined over statements in a block, however the analysis is still relevant in a functional setting. For composition of single-parameter functions, liveness for the parameter of the composed function is defined by the liveness of the individual function parameters as follows:

$g : B \to C$	$f : A \to B$	$g \circ f : A \to C$	
discarded	*discarded*	*discarded*	} g discards result of f $\therefore$ liveness of f irrelevant
discarded	*live*	*discarded*	
live	*discarded*	*discarded*	} g needs result of f $\therefore$ liveness of $g \circ f$ is liveness of f
live	*live*	*live*	

The liveness of composition is therefore the *conjunction* of the liveness properties for the composed functions, where *live* is true and *discarded* is false.

An *approximate* liveness analysis is described by a coeffect system, with coeffects taken from the set $\{\mathsf{F}, \mathsf{T}\}$ where F denotes that context of free-variables is *definitely not live*, and T that the context of free-variables *may be live*. This set of liveness coeffects is ordered with $\mathsf{F} \sqsubseteq \mathsf{T}$ since any variables that are definitely not live may be safely approximated as live. **Figure** 6.3 gives the rules of the liveness coeffect system.

The base cases of constants [CONST] and variables [VAR] are assigned the two possible coeffects F and T respectively, since constants discard the context whilst variables use the context. The coeffects of *let*-binding [LET] are those of the receiving term $(\Gamma, v : \tau' \,?\, r \vdash e_2 : \tau)$ because a discarded context for e_2 implies that v is discarded thus the bound expression e_1 is dead and its liveness is irrelevant. A live context for e_2 implies either Γ or v is live, but it cannot be known which, thus the context Γ of the resulting *let*-expression must be live.

The [ABS] rule splits the coeffect of the function body between a function's defining context (coeffect m) and application context (latent coeffect n) where the coeffect of the function body

$$[\textsc{const}]\frac{}{\Gamma \,?\, \mathsf{F} \vdash c : \tau} \qquad [\textsc{var}]\frac{v : \tau \in \Gamma}{\Gamma \,?\, \mathsf{T} \vdash v : \tau}$$

$$[\textsc{abs}]\frac{\Gamma, v : \sigma \,?\, m \wedge n \vdash e : \tau}{\Gamma \,?m \vdash \lambda v.e : \sigma \xrightarrow{n} \tau} \qquad [\textsc{app}]\frac{\Gamma \,?\, n \vdash e_1 : \sigma \xrightarrow{p} \tau \quad \Gamma \,?\, m \vdash e_2 : \sigma}{\Gamma \,?\, n \vee (p \wedge m) \vdash e_1\ e_2 : \tau}$$

$$[\textsc{let}]\frac{\Gamma \,?\, s \vdash e_1 : \tau' \quad \Gamma, v : \tau' \,?\, r \vdash e_2 : \tau}{\Gamma \,?\, r \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \tau} \qquad [\textsc{if}]\frac{\Gamma \,?\, r \vdash e : \mathsf{bool} \quad \Gamma \,?\, s \vdash e_1 : \tau \quad \Gamma \,?\, t \vdash e_2 : \tau}{\Gamma \,!\, r \vee s \vee t \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

Figure 6.3. Type and coeffect system for liveness

equals $m \wedge n$. Thus, if the context of the function body is live then both Γ and v are live. If the context of the body is discarded then a kind of sub-coeffecting may occur, where Γ and v are either discarded or one is live. The [ABS] rule introduces a source of non-determinism, where the coeffect judgment for an abstraction may be non-unique (*i.e.*, non-*principal*). Non-unique coeffects are discussed further in Section 6.2.3.

For application [APP], if the context of the function subexpression is live then the context of application is live. However, if the context of the function is discarded, liveness of the application is equivalent to the sequential composition of the function with the argument, and thus the conjunction of their respective coeffects is taken. The operation $\vee$ therefore combines coeffects of multiple subterms which share the same free-variable context. This is used in the [IF] rule where the context of three subterms are combined disjunctively.

The semantics of this language can be captured by the *indexed partiality comonad* (Example 5.4.2, p. 105) with some additional structure which will be discussed later.

6.2.2. Dataflow

Consider a dataflow language in the style of Lucid. For simplicity, the language will allow only *causal* behaviour, where computations may depend on the present and past values of variables. The syntax is of the λ-calculus with a keyword prev accessing the "previous" value of a computation (which can be applied repeatedly to access further past values). The semantics is modelled by causal streams (Section 3.2.4).

A coeffect analysis is defined here which tracks how far into the past values are accessed, or equivalently, tracking data access on streams, *i.e.*, how many items in a stream's past are accessed. This analysis allows the efficient caching of stream elements in the semantics based on their data access pattern. Coeffects are natural numbers $\mathbb{N}$ where a coeffect $n \in \mathbb{N}$ means that computing an element of the result stream requires up to n-elements previous to the current element from streams in the context.

Consider two coKleisli stream functions $f : \mathsf{Stream}\, A \to B, g : \mathsf{Stream}\, B \to C$ respectively requiring m and n previous elements from their parameter streams to compute a single output element. The composition $g \mathbin{\hat{\circ}} f : \mathsf{Stream}\ A \to C$, requires only the minimal number of previous

$$[\textsc{const}]\frac{}{\Gamma\,?\,0 \vdash c : \tau} \qquad [\textsc{var}]\frac{v : \tau \in \Gamma}{\Gamma\,?\,0 \vdash v : \tau} \qquad [\textsc{prev}]\frac{\Gamma\,?\,n \vdash e : \tau}{\Gamma\,?\,n+1 \vdash \mathsf{prev}\ e : \tau}$$

$$[\textsc{abs}]\frac{\Gamma, v : \sigma\,?\,s\ \mathsf{min}\ t \vdash e : \tau}{\Gamma\,?s \vdash \lambda v.e : \sigma \xrightarrow{t} \tau} \qquad [\textsc{app}]\frac{\Gamma\,?\,r \vdash e_1 : \sigma \xrightarrow{s} \tau \quad \Gamma\,?\,t \vdash e_2 : \sigma}{\Gamma\,?\,r\ \mathsf{max}\ (s+t) \vdash e_1\ e_2 : \tau}$$

$$[\textsc{let}]\frac{\Gamma\,?\,s \vdash e_1 : \tau' \quad (\Gamma, v : \tau')\,?\,r \vdash e_2 : \tau}{\Gamma\,?\,r+s \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \tau} \qquad [\textsc{if}]\frac{\Gamma\,?\,r \vdash e : \mathsf{bool} \quad \Gamma\,?\,s \vdash e_1 : \tau \quad \Gamma\,?\,t \vdash e_2 : \tau}{\Gamma\,?\,r\ \mathsf{max}\ s\ \mathsf{max}\ t \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

Figure 6.4. Type and coeffect system for causal dataflow

input elements, and can be defined as follows:

$$\lambda a.\, g\,\langle f\langle a_0, \ldots, a_m\rangle, \ldots, f\langle a_n, \ldots, a_{n+m}\rangle\rangle$$

thus the composite function requires $n+m$ previous elements from a (similarly to the composition of *partial local operations* on lists defined in Section 5.2, p. 99). It is safe to provide more elements than required, since extra elements are ignored. Coeffects are therefore ordered by $\leqslant$.

Figure 6.4 summarises the coeffect analysis for causal dataflow. Note that as with the liveness analysis, coeffects are defined over the whole context.

Constants and variables both require no previous elements from streams in the context, whilst prev requires one more element. The [LET] rule shows that **let**-binding has the same coeffects as the composition described above, where the receiving term e_2 may require r previous elements from either Γ or x (but due to the overapproximation it is not known which). Thus s elements are required for each element of e_1 passed to e_2, thus $r + s$ elements in total.

For abstraction [ABS], the coeffect of the defining context and application context are such that their minimum matches the coeffect of the function body. Thus, both the defining and application context must provide at least enough elements to satisfy the function bodies requirements (in a bottom-up reading). For application, r previous elements are required to compute a single function value of e_1 where its application to e_2 requires then $s + t$ previous elements from the context, thus [APP] takes the maximum of r and $s + t$. Therefore, when two contexts are combined the maximum number of previous elements is taken, as used in [IF].

The semantics of this language is based on the indexed list comonad (discussed later).

6.2.3. Dynamically-bound resources

This chapter's introduction used the example of a language with resources where coeffects track resource usage. This example is extended here to dynamically-bound resources in a distributed setting. Different resources may be available in different execution contexts, where a computation may have a resource identifier rebound to a local version of the resource.

Consider the following expression which accesses a database (db) and clock resource (clk):

```
query (access db) "select * from table where date > %1" (access clk)        (50)
```

$$[\textsc{const}]\frac{}{\Gamma\,?\,\emptyset \vdash c : \tau} \qquad [\textsc{var}]\frac{v : \tau \in \Gamma}{\Gamma\,?\,\emptyset \vdash v : \tau} \qquad [\textsc{access}]\frac{(r,\tau) : \mathbb{R}}{\Gamma\,?\,\{r : \tau\} \vdash \mathsf{access}\, r : \tau}$$

$$[\textsc{abs}]\frac{(\Gamma, v : \sigma)\,?\,s \cup t \vdash e : \tau}{\Gamma\,?s \vdash \lambda v.e : \sigma \xrightarrow{t} \tau} \qquad [\textsc{app}]\frac{\Gamma\,?\,s \vdash e_1 : \sigma \xrightarrow{t} \tau \qquad \Gamma\,?\,r \vdash e_2 : \sigma}{\Gamma\,?\,r \cup s \cup t \vdash e_1\, e_2 : \tau}$$

$$[\textsc{let}]\frac{\Gamma\,?\,s \vdash e_1 : \tau' \qquad \Gamma, x : \tau'\,?\,r \vdash e_2 : \tau}{\Gamma\,?\,r \cup s \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau} \qquad [\textsc{if}]\frac{\Gamma\,?\,r \vdash e : \mathsf{bool} \qquad \Gamma\,?\,s \vdash e_1 : \tau \qquad \Gamma\,?\,t \vdash e_2 : \tau}{\Gamma\,!\,r \cup s \cup t \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

Figure 6.5. Type and coeffect system for resources

This program may be executed on a server where the db and clk resources are locally bound to the server's database and clock. Alternatively, the program may be executed on a client device where db is bound to the server's database and clk is rebound to the local clock.

Figure 6.5 provides a coeffect analysis tracking the set of resources which *may be accessed* by an expression; $\mathbb{R}$ is a set of pairs of possible resource identifiers (atoms) and their types. The resources used by an expression can always be overapproximated, thus coeffects are ordered by subset ordering $\subseteq$. Both constants and variables have the empty coeffect since they do not access any resources. For access, the coeffect is a singleton set of the accessed resource. The [LET], [IF] and [APP] rules compose the coeffects of their subterms by the union $\cup$. Abstraction [ABS] allows resource access from either the defining context or application context.

For the above example (50), there are nine possible coeffect inferences, of which three are:

$$\begin{aligned} \Gamma\,?\,\emptyset \vdash \ldots &: \sigma \xrightarrow{\{\mathsf{db},\mathsf{clk}\}} \tau \\ \Gamma\,?\,\{\mathsf{db},\mathsf{clk}\} \vdash \ldots &: \sigma \xrightarrow{\{\mathsf{clk}\}} \tau \\ \Gamma\,?\,\{\mathsf{db}\} \vdash \ldots &: \sigma \xrightarrow{\{\mathsf{clk}\}} \tau \end{aligned}$$

Depending on the purpose of the analysis, uniqueness of coeffects can be enforced by language constructs which reduce the non-determinism by requiring particular coeffects of their subterms, or, by the compiler which might require particular top-level coeffects.

In the distributed programming setting here, the language may provide a construct for *serialising* and *transmitting* a function to another device, *e.g.*:

$$[\textsc{send}]\frac{\Gamma\,?\,R \vdash e : \sigma \xrightarrow{S} \tau \qquad S \subseteq \mathit{resources}(\mathit{device})}{\Gamma\,?\,R \vdash \mathsf{send}\ \mathit{device}\ e\ \mathsf{with}\ [S] : ()}$$

where $\mathit{resources}(\mathit{device})$ is the set of resources for a device, and send ... with ... describes the resources S which are to be rebound by the receiving device. Thus [SEND] resolves how to split the coeffects of a function. The exact outer coeffect for R here could be chosen by the compiler or runtime system, matching R with the available resources on the host device.

For the *liveness* and *dataflow* examples, which provide *optimising semantics*, non-determinism is less problematic. The compiler could resolve non-determinism by selecting the *least* coeffect *i.e.*, the least contextual requirements and therefore the most efficient implementation.

$$[\textsc{var}]\ \frac{v : \tau \in \Gamma}{\Gamma \,?\, I \vdash v : \tau} \qquad [\textsc{abs}]\ \frac{\Gamma, v : \sigma \,?\, s \sqcap t \vdash e : \tau}{\Gamma \,?\, s \vdash \lambda v.e : \sigma \xrightarrow{t} \tau} \qquad [\textsc{sub}]\ \frac{\Gamma \,?\, s \vdash e : \tau \quad s \sqsubseteq s'}{\Gamma \,?\, s' \vdash e : \tau}$$

$$[\textsc{let}]\ \frac{\Gamma \,?\, s \vdash e_1 : \tau' \quad \Gamma, v : \tau' \,?\, r \vdash e_2 : \tau}{\Gamma \,?\, r \sqcup (r \bullet s) \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \tau} \qquad [\textsc{app}]\ \frac{\Gamma \,?\, r \vdash e_1 : \sigma \xrightarrow{s} \tau \quad \Gamma \,?\, t \vdash e_2 : \sigma}{\Gamma \,?\, r \sqcup (s \bullet t) \vdash e_1\ e_2 : \tau}$$

Figure 6.6. General coeffect system for the coeffect calculus

	C	$\bullet$	I	$\sqcup$	$\sqcap$	$\sqsubseteq$
liveness	$\mathbb{B}$	$\wedge$	T	$\vee$	$\wedge$	$\mathsf{F} \sqsubseteq \mathsf{T}$
dataflow	$\mathbb{N}$	$+$	0	max	min	$\leqslant$
resources	$\mathcal{P}(\mathbb{R})$	$\cup$	$\emptyset$	$\cup$	$\cup$	$\subseteq$

Figure 6.7. Instances of coeffect algebras for the example languages of Section 6.2.

6.3. The general calculus

The three example languages introduced in the previous section have a common structure to their semantics, which is exposed by the common structure of their coeffect analyses. These examples are all instances of the general *coeffect calculus*. The general calculus has the syntax and typing of the simply-typed λ-calculus with *let*-binding, and provides a general semantics of contextual computations which is described by a general coeffect system. **Figure** 6.6 shows the calculus and the coeffect system which is parameterised by an instance of a *coeffect algebra* for the particular notion of context.

Definition 6.3.1. A *coeffect algebra* $(C, \bullet, I, \sqcup, \sqcap)$ comprises a set of coeffects C and:

- $(C, \bullet, I)$ – a monoid for the coeffect of composition ([LET], [APP], I for [VAR]);
- $(C, \sqcup)$ – a semigroup for combining coeffects of subterms (shared contexts) ([LET], [APP]), inducing a pre-order $\sqsubseteq$ where $X \sqsubseteq Y$ iff $X \sqcup Y = Y$.
- $(C, \sqcap)$ – a semigroup for combining coeffects of the definition context and application context for a function ([ABS]).
- axiom – distributivity of $\bullet$ over $\sqcup$:

$$X \bullet (Y \sqcup Z) \equiv (X \bullet Y) \sqcup (X \bullet Z) \qquad (Y \sqcup Z) \bullet X \equiv (Y \bullet X) \sqcup (Z \bullet X)$$

i.e., $\bullet$ is monotonic with respect to the preorder $\sqsubseteq$ induced by $\sqcup$.

The coeffect calculus (**Figure** 6.6) therefore captures the general propagation of contextual requirements through a program. The coeffect algebra structure provides flexibility with separate coeffect operations for composition ($\bullet$), abstraction ($\sqcap$), and sharing ($\sqcup$). **Figure** 6.7 summarises the operations used in the coeffect systems of the previous section as coeffect algebras. For resources, $\mathbb{R}$ is the set of resource identifiers paired with their types.

	$\sqcup$	$\bullet$	$r \sqcup (r \bullet s)$
liveness	$\vee$	$\wedge$	$r = r \vee (r \wedge s)$
dataflow	max	$+$	$r + s = r \mathsf{\ max\ } (r + s)$
resources	$\cup$	$\cup$	$r \cup s = r \cup (r \cup s)$

Figure 6.8. Comparison of [LET] for the example languages of Section 6.2.

Let-binding. The coeffect of [LET] in the general calculus appears more complex than the coeffects of [LET] in the example languages. The coeffect of [LET] in each of the examples is equivalent to the more complex coeffect term here by additional properties of each coeffect algebras, summarised in **Figure** 6.8.

Structural rules. The general coeffect calculus admits the usual structural rules of weakening, exchange, and contraction which preserve the coeffects of their premises in the conclusion.

6.3.1. Syntactic equational theory

The equality relation on terms for the simply-typed λ-calculus can now be extended such that two terms are equal if they have the same context, type, and coeffect, *i.e.*, $\Gamma ? R \vdash e_1 \equiv e_2 : \tau$. This section considers potential equations on terms of the coeffect calculus (following the usual equational theory of the simply-typed λ-calculus with *let*-binding, Section 2.1.2, p. 22) and any additional coeffect algebra axioms required for these equalities; for an equality $\Gamma \vdash e_1 \equiv e_2 : \tau$ in the simply-typed λ-calculus, where $\Gamma ? R \vdash e_1 : \tau$ and $\Gamma ? S \vdash e_2 : \tau$ are judgments in the coeffect calculus, this section describes the additional coeffect algebra axioms such that $R \equiv S$.

So far, the coeffect algebra axioms (monoid and semigroup axioms) have been minimal such that the general calculus is flexible, without committing to any particular equational theories.

6.3.1.1. Let-binding

In the pure simply-typed λ-calculus with *let*-binding, **let** has associativity, left unit, right unit, weakening, contraction, and exchange properties (Section 2.2.2, p. 28). The following considers each equation, showing the type-and-coeffect judgment of the equation and the required axioms on the coeffect algebra. Appendix B.4.1 (p. 191) shows the full type-and-coeffect derivations for the equations here. Axioms already included in the coeffect algebra definition are marked with ▸. Additional axioms that are not part of the coeffect algebra definition are marked with $+$.

Associativity.

$$\begin{aligned} & \Gamma ? ((Z \sqcup (Z \bullet Y)) \sqcup ((Z \sqcup (Z \bullet Y)) \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ (\mathbf{let}\ y = e_2\ \mathbf{in}\ e_3) : \tau_3 \\ \equiv\ & \Gamma ? (Z \sqcup (Z \bullet (Y \sqcup (Y \bullet X)))) \vdash \mathbf{let}\ y = (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)\ \mathbf{in}\ e_3 : \tau_3 \end{aligned}$$

Equality of the two coeffect terms for *let*-binding associativity follows from:

- ▸ associativity of $\sqcup$ (from semigroup)
- ▸ associativity of $\bullet$ (from monoid)
- ▸ distributivity of $\bullet$ over $\sqcup$ (*i.e.*, $X \bullet (Y \sqcup Z) = (X \bullet Y) \sqcup (X \bullet Z)$)

+ distributivity of $\sqcup$ over $\bullet$, *i.e.*, $X \sqcup (Y \bullet Z) = (X \sqcup Y) \bullet (X \sqcup Z)$
  or I is the (left) unit of $\sqcup$

These axioms hold for all the example coeffect algebra, where for resources and liveness $\sqcup$ distributes over $\bullet$ and for dataflow the alternate axiom that $I \sqcup X = X$ holds ($0 \,\mathsf{max}\, X = X$).

Left unit.

$$\Gamma \,?\, (I \sqcup (I \bullet X)) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ x : \tau_1 \quad \equiv \quad \Gamma \,?\, X \vdash e_1 : \tau_1$$

Equality of the two coeffect terms for a left unit property of **let** follows from:

- ▸ I the unit of $\bullet$ (monoid)
- + I the unit of $\sqcup$ (*i.e.*, $I \sqcup Y = Y$, that is, I is the least element and $\sqcup$ is the least-upper bound **or** I is the greatest element and $\sqcup$ is the greatest-lower bound).

The additional axioms holds for dataflow ($0 \,\mathsf{max}\, X = X$) and resources ($\emptyset \cup X = X$). For liveness, this does not hold (since, $\mathsf{T} \vee X = \mathsf{T}$). This is a consequence of the imprecision of liveness coeffects (where the coeffect system approximates liveness over the whole context, thus where x is live the whole context is live).

Right unit.

$$\Gamma, y : \tau_1 \,?\, (Y \sqcup (Y \bullet I)) \vdash \textbf{let}\ x = y\ \textbf{in}\ e_2 : \tau_2 \quad \equiv \quad \Gamma, y : \tau_1 \,?\, Y \vdash e_2[x := y] : \tau_2$$

Equality of the coeffect terms follows from:

- ▸ I the unit of $\bullet$ (monoid)
- + $\sqcup$ is idempotent ($Y \sqcup Y = Y$)

These hold for all the example calculi here. This rule corresponds to α-renaming via **let**-binding.

Weakening.

$$\Gamma \,?\, (Y \sqcup (Y \bullet X)) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2 \quad \equiv \quad \Gamma \,?\, Y \vdash e_2 : \tau_2 \qquad (x \notin FV(e_2))$$

This rule highlights the call-by-value nature of *let*-binding in the coeffect calculus, where coeffects are eagerly applied. The two coeffect terms are equal if:

- + $\bullet$ is inversely monotonic to $\sqcup$, *i.e.*, either:
  1. $\bullet$ is decreasing and $\sqcup$ is the least-upper bound;
  2. $\bullet$ is increasing and $\sqcup$ is the greatest-lower bound (despite the suggestive notation).

This additional axiom holds only for the liveness coeffect calculus with $\bullet = \wedge$ which is monotonically decreasing and $\sqcup = \vee$ which is the least-upper bound.

Contraction.

$$\begin{aligned} & \Gamma \,?\, ((Z \sqcup (Z \bullet X)) \sqcup ((Z \sqcup (Z \bullet X)) \bullet X)) \vdash \textbf{let}\ x = e_1\ \textbf{in}\ (\textbf{let}\ y = e_1\ \textbf{in}\ e_2) : \tau_2 \\ \equiv \quad & \Gamma \,?\, (Z \sqcup (Z \bullet X)) \vdash \textbf{let}\ z = e_1\ \textbf{in}\ e_2[z := y][z := x] : \tau_2 \end{aligned}$$

Equality of the two coeffect terms follows from:

- ▸ monoidality of $(C, \bullet, I)$
- ▸ distributivity of $\bullet$ over $\sqcup$
- \+ idempotence of $\bullet$

Only liveness and resources have an idempotent $\bullet$; dataflow does not permit contraction.

Exchange.

$$\begin{aligned} & \Gamma \,?\, ((Z \sqcup (Z \bullet Y)) \sqcup ((Z \sqcup (Z \bullet Y)) \bullet X)) \vdash \textbf{let } x = e_1 \textbf{ in } (\textbf{let } y = e_2 \textbf{ in } e_3) : \tau_3 \\ \equiv \quad & \Gamma \,?\, ((Z \sqcup (Z \bullet X)) \sqcup ((Z \sqcup (Z \bullet X)) \bullet Y)) \vdash \textbf{let } y = e_2 \textbf{ in } (\textbf{let } x = e_1 \textbf{ in } e_3) : \tau_3 \end{aligned}$$

Equality of the two coeffect terms follows from:

- ▸ monoidality of $(C, \bullet, I)$
- ▸ distributivity of $\bullet$ over $\sqcup$
- \+ commutative $\bullet$

These axioms hold in all of the example coeffect algebra here.

6.3.1.2. $\beta\eta$-equality

Let-binding and abstraction-application equality. Recall [$\equiv$-*let*-λ] for the simply-typed λ-calculus (Section 2.1.2, p. 22), *i.e.*, $\textbf{let } v = e_1 \textbf{ in } e_2 \equiv (\lambda v.e_2)e_1$.

In the coeffect calculus, [$\equiv$-*let*-λ] has the type/coeffect derivations:

$$\frac{\Gamma \,?\, Z \vdash e_1 : \tau_1 \qquad \Gamma, v : \tau_1 \,?\, (X \sqcap Y) \vdash e_2 : \tau_2}{\Gamma \,?\, ((X \sqcap Y) \sqcup ((X \sqcap Y) \bullet Z)) \vdash \textbf{let } v = e_1 \textbf{ in } e_2 : \tau_2} \equiv \frac{\dfrac{\Gamma, v : \tau_1 \,?\, (X \sqcap Y) \vdash e_2 : \tau_2}{\Gamma \,?\, X \vdash (\lambda v.e_2) : \tau_1 \xrightarrow{Y} \tau_2} \qquad \Gamma \,?\, Z \vdash e_1 : \tau_1}{\Gamma \,?\, (X \sqcup (Y \bullet Z)) \vdash (\lambda v.e_2)\, e_1 : \tau_2}$$

Equality of the two coeffect terms here follows from:

- ▸ monoidality of $(C, \bullet, I)$
- \+ idempotence of $\sqcup$
- \+ an *interchange law* between $\bullet$ and $\sqcup$, *i.e.*, $(X \bullet Y) \sqcup (Z \bullet W) \equiv (X \sqcup Z) \bullet (Y \sqcup W)$
- \+ equivalence of operations $\sqcap = \sqcup$

The last axiom is particularly strong; these conditions hold only for the resources calculus.

Alternatives. An alternate [$\equiv$-*let*-λ] rule holds only when $\sqcap$ is idempotent, with the following:

$$[\textsc{let}]\ \frac{\Gamma \,?\, S \vdash e_1 : \sigma \qquad \Gamma, v : \sigma \,?\, R \vdash e_2 : \tau}{\Gamma \,?\, R \sqcup (R \bullet S) \vdash \textbf{let } v = e_1 \textbf{ in } e_2 : \tau} \equiv [\textsc{app}]\ \frac{\Gamma \,?\, S \vdash e_1 : \sigma \qquad [\textsc{abs}]\ \dfrac{\Gamma, v : \sigma \,?\, R \vdash e_2 : \tau}{\Gamma \,?\, R \vdash \lambda v.e_2 : \sigma \xrightarrow{R} \tau}}{\Gamma \,?\, R \sqcup (R \bullet S) \vdash (\lambda v.e_2)e_1 : \tau}$$

Note that the premises of these two [$\equiv$-*let*-λ] rules differ: the first is more general, the second applies only in cases where [ABS] duplicates the coeffects of its function body. This restricted form of [$\equiv$-*let*-λ] holds for all of the example calculi here (*i.e.*, $\sqcap$ idempotent in all calculi).

A more general **let** may be provided which is equivalent to abstraction followed by application where the coeffects of the body are split, of the form:

$$[\textsc{let}]\frac{\Gamma \,?\, S \vdash e_1 : \sigma \qquad \Gamma, v : \sigma \,?\, R_1 \sqcap R_2 \vdash e_2 : \tau}{\Gamma \,?\, R_1 \sqcup (R_2 \bullet S) \vdash \mathbf{let}\ v = e_1\ \mathbf{in}\ e_2 : \tau}$$

The original form of *let*-binding is however preferred, as its propagation of coeffects is more intuitive, combining a single coeffect term from both the substituting and receiving terms rather than splitting the coeffects for the receiving term (R_1 and R_2 above).

β-equality. Given [$\equiv$-*let*-λ] holds, β-equality holds if *let*-binding is equivalent to substitution ([*let*-β]). A meta-level substitution lemma is therefore needed to give the coeffects of syntactic substitution, which matches the coeffects of *let*-binding. For the general coeffect calculus, the following substitution lemma holds given a *substituting coeffect algebra* (described below):

Lemma 6.3.2 (Substitution). *Given a* substituting *coeffect algebra* C, $\Gamma, x : \tau' \,?\, R \vdash e : \tau$, *and* $\Gamma \,?\, S \vdash e' : \tau'$ *then* $\Gamma \,?\, R \sqcup (R \bullet S) \vdash e[x := e'] : \tau$.

Definition 6.3.3. A *substituting coeffect algebra* is a coeffect algebra over a set C with the additional axioms that:

+ I is either the least or greatest element of $\sqsubseteq$, *i.e.*, $\forall X.I \sqsubseteq X$ or $\forall X.X \sqsubseteq I$
+ $(C, \sqcap, I)$ is a monoid
+ *interchange* between $\bullet$ and $\sqcap$, *i.e.*, $(X \bullet Y) \sqcap (Z \bullet W) \equiv (X \sqcap Z) \bullet (Y \sqcap W)$.

Appendix C.4.2 (p. 204) provides the (inductive) proof which elicits these additional axioms. The first axiom is required for the [VAR] case of substitution, the second axiom for [APP] and [ABS], and the third and fourth axioms for [ABS]. Furthermore, the proof uses the monoidal properties of $(C, \bullet, I)$ and distributivity of $\bullet$ over $\sqcup$.

Subsequently, β-equality holds (syntactically) given a substituting coeffect algebra:

$$\begin{array}{l}[\textsc{abs}]\ \dfrac{\Gamma, x : \sigma \,?\, R \vdash e : \tau}{\Gamma \,?\, R \vdash \lambda x\,.\,e : \sigma \xrightarrow{R} \tau} \qquad \Gamma \,?\, S \vdash e' : \sigma \\ [\textsc{app}]\ \dfrac{}{\Gamma \,?\, R \sqcup (R \bullet S) \vdash (\lambda x\,.\,e)\,e' : \tau}\end{array} \quad \equiv \quad \Gamma \,?\, R \sqcup (R \bullet S) \vdash e[x := e'] : \tau'$$

Thus, *subject reduction* also has the same conditions.

Lemma 6.3.4 (Subject reduction). *For a substituting coeffect algebra, given* $\Gamma \,?\, R \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \,?\, R \vdash e' : \tau$.

Both the liveness and resources coeffect algebras are instances of substituting coeffect algebras; dataflow is not. Therefore, β-equality holds only in the resource and liveness examples. The situation with dataflow can be compared with stateful computations where β-equality does not hold since side effects may be duplicated. Instead, a restricted β-equality could be provided for values, where arguments to a function are evaluated first by *let*-binding.

η-equality. For a term $\Gamma \,?\, R \vdash e : \sigma \xrightarrow{S} \tau$, η-equality is respected by a coeffect calculus if $\Gamma \,?\, R \vdash e \equiv (\lambda x.e\ x) : \sigma \xrightarrow{S} \tau$ (and $x \notin FV(e)$). The type/coeffect derivations for η-equality are:

$$[\textsc{abs}]\ \frac{[\textsc{app}]\ \dfrac{[\textsc{weaken}]\ \dfrac{\Gamma \,?\, R \vdash e : \sigma \xrightarrow{S} \tau}{\Gamma, x : \sigma \,?\, R \vdash e : \sigma \xrightarrow{S} \tau} \qquad \Gamma, x : \sigma \,?\, I \vdash x : \sigma}{\Gamma, x : \sigma \,?\, R \sqcup (S \bullet I) \vdash e\,x : \tau}}{\Gamma \,?\, P \vdash \lambda x.e\,x : \sigma \xrightarrow{Q} \tau} \qquad \equiv\ \Gamma \,?\, R \vdash e : \sigma \xrightarrow{S} \tau$$

where $R \sqcup (S \bullet I) = P \sqcap Q$. Therefore η-equality holds only when $P \sqcap Q = R \sqcup S$ and $R = P$ (for the coeffect terms to be equal) and $S = Q$ (for the types to be equal), therefore $R \sqcap S = R \sqcup S$. This property holds only of the resources example here (where $\sqcup = \sqcap = \cup$).

Therefore, full $\beta\eta$-equality holds for a substituting coeffect algebra with $\sqcup = \sqcap$.

6.3.1.3. Summary

Figure 6.9 summarises which equations hold for the three example coeffect calculi of resources, liveness, and dataflow in this chapter.

Note that in all of the examples here, there are some additional axioms common to all examples that are not part of the coeffect algebra definition: commutativity of $\bullet$, idempotence of $\sqcup$, and idempotence of $\sqcap$. These imply that, in the example calculi, *let*-binding is associative, has the right unit and exchange properties, and $\mathbf{let}\, x = e_1\, \mathbf{in}\, e_2 \equiv (\lambda x.e_2)\, e_1$ in the restricted case of duplicating the coeffects of e_2 for the latent and immediate coeffects of the abstraction. Although these axioms hold in all the examples here they are not part of the coeffect algebra definition for the sake of generality. Further work is to find useful examples which do not have these properties to assess if this choice is justified.

6.3.2. Additional constructions

Constants. The general calculus purposefully omits constants to keep the calculus as general as possible, since constants are not necessarily part of the simply-typed λ-calculus. However,

		resources	liveness	dataflow
let	associativity	✓	✓	✓
	left unit	✓		✓
	right unit	✓	✓	✓
	weakening		✓	
	contraction	✓	✓	
	exchange	✓	✓	✓
$\beta\eta$	[$\equiv$-*let*-λ]	✓		
	[$\equiv$-*let*-λ] (idempotence)	✓	✓	✓
	[$\equiv$-β]	✓	✓	
	[$\equiv$-η]	✓		

Figure 6.9. Summary of term equalities valid for the example coeffect calculi of this chapter.

constants can be added with a coeffect $\bot \in C$ and the coeffect rule:

$$[\textsc{const}]\frac{c : T}{\Gamma ? \bot \vdash c : T}$$

where $(C, \sqcup, \bot)$ is a bounded semi-lattice. For the examples here, these correspond to the following:

	C	$\sqcup$	$\bot$
liveness	$\mathbb{B}$	$\vee$	F
dataflow	$\mathbb{N}$	max	0
resources	$\mathcal{P}(\mathbb{R})$	$\cup$	$\emptyset$

Note that it is not necessarily the case that $\bot \sqcap R = \bot$, *i.e.*, $\bot$ is not necessarily the least element for $\sqcap$ (for example, in resource calculus $\emptyset \cup R = R$).

Tuples. Tuple types can be added to the coeffect calculus using $\sqcup$ for their coeffects:

$$[\textsc{tup}]\frac{\Gamma \,?\, R \vdash e_1 : \tau_1 \qquad \Gamma \,?\, S \vdash e_2 : \tau_2}{\Gamma \,?\, R \sqcup S \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

The empty tuple constant, or *unit type*, the may also be included with $\Gamma \,?\, I \vdash () : ()$.

Conditionals. Conditional statements can be added to general coeffect calculus using $\sqcup$:

$$[\textsc{if}]\frac{\Gamma \,?\, R \vdash e : \mathsf{bool} \qquad \Gamma \,?\, S \vdash e_1 : \tau \qquad \Gamma \,?\, T \vdash e_2 : \tau}{\Gamma \,!\, R \sqcup S \sqcup T \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : \tau}$$

6.4. Categorical semantics

Chapter 5 introduced *indexed comonads* to capture a more precise notion of context-dependent computation than regular comonads. Indexed comonads provide contexts that are indexed by a computation's contextual requirements, *i.e.*, they are essentially indexed by a coeffect. Indexed comonads are used here as the basis for the categorical semantics of the general coeffect calculus.

An initial semantics for the coeffect λ-calculus is systematically constructed following the approach of Chapter 2 and by analogy with the semantics of the contextual λ-calculus in Chapter 3. This semantics is shown first in Section 6.4.1, which gives a semantics that *approximates* the behaviour described by the general coeffect system, that is, its morphisms require contexts which are more general than that described by the coeffect system in **Figure** 6.6. Section 6.4.2 refines the semantics such that it corresponds exactly to the general coeffect system described above, and relates the two approaches.

Section 6.4.3 considers additional axioms required on the mathematical structure behind the semantics for the semantics to exhibit $\beta\eta$-equality. Section 6.4.4 instantiates the semantics for the liveness, resources, and dataflow coeffect calculi shown in this chapter.

Categorical domain. The semantics of the general coeffect calculus, with coeffect algebra $(C, \bullet, I, \sqcup, \sqcap)$, requires an indexed comonad $\mathsf{D} : \mathbb{I}^{\mathsf{op}} \to [\mathbb{C}, \mathbb{C}]$ where $\mathbb{I}$ is the category of the partial order $(C, \sqsubseteq)$ *i.e.* $\mathbb{I}_0 = \mathbb{I}_0^{\mathsf{op}} = C$ and for all $X, Y \in C$, if $X \sqsubseteq Y$ then there is a unique morphism $Y \to X \in \mathbb{I}_1^{\mathsf{op}}$ (and $X \to Y \in \mathbb{I}_1$). $(\mathbb{I}^{\mathsf{op}}, \bullet, I)$ forms a *monoidal category*.

Proposition 6.4.1. *For a coeffect algebra* $(C, \bullet, I, \sqcup, \sqcap)$ *where* $\mathbb{I}$ *is the category of the partial order* $(C, \sqsubseteq)$*, then* $(\mathbb{I}^{\mathsf{op}}, \bullet, I)$ *forms a* monoidal category *with bifunctor* $\bullet : \mathbb{I}^{\mathsf{op}} \times \mathbb{I}^{\mathsf{op}} \to \mathbb{I}^{\mathsf{op}}$*.*

Proof. (sketch) Given $f : X \to Y, g : A \to B \in \mathbb{I}_1^{\mathsf{op}}$ then $f \bullet g : X \bullet A \to Y \bullet B \in \mathbb{I}_1^{\mathsf{op}}$ and therefore $Y \sqsubseteq X$ and $B \sqsubseteq A$ implies $Y \bullet B \sqsubseteq X \bullet A$, *i.e.*, $\bullet$ is monotonic (equivalently, bifunctorial), the proof of which is in Appendix C.4.1.1 (p. 204). The associativity and unital properties of the monoidal category (Appendix B.1.2, p. 185) follow from the monoidality of $(C, \bullet, I)$. □

The semantics maps type-and-coeffect judgments to morphisms of the *indexed coKleisli category* $\mathbb{C}_{\mathsf{D}}$ for D, which has objects $\mathbb{C}_{\mathsf{D}0} = \mathbb{C}_0$ and morphisms $\mathbb{C}_{\mathsf{D}}(A, B) = \bigcup_{R \in \mathbb{I}_0} \mathbb{C}(\mathsf{D}_R A, B)$, where:

$$[\![\Gamma ? R \vdash e : \tau]\!] : \mathsf{D}_R [\![\Gamma]\!] \to [\![\tau]\!] \in \mathbb{C}_1$$

The mapping $[\![-]\!]$ is defined in the usual way for free-variable typing contexts. For types, $[\![-]\!]$ maps function types with latent coeffect requirements to hom-objects as follows:

$$[\![\sigma \xrightarrow{R} \tau]\!] = (\mathsf{D}_R [\![\sigma]\!] \Rightarrow [\![\tau]\!])$$

6.4.1. Derived semantics

A semantics for the coeffect calculus can be systematically constructed following the approach of Chapter 2 which was used for the semantics of the contextual λ-calculus in Chapter 3.

As shown in Section 3.3.1 for the contextual λ-calculus, a coKleisli category requires a lax and colax semi-monoidal functor with $\mathsf{m}_{A,B} : \mathsf{D}A \times \mathsf{D}B \to \mathsf{D}(A \times B)$ and $\mathsf{n}_{A,B} : \mathsf{D}(A \times B) \to \mathsf{D}A \times \mathsf{D}B$ to provide curry/uncurry operations. Analogous natural transformations for the indexed setting are as follows, where $\sqcup, \sqcap : \mathbb{I}_0 \times \mathbb{I}_0 \to \mathbb{I}_0$ are object mappings:

$$\mathsf{m}^{R,S}_{A,B} : \mathsf{D}_R A \times \mathsf{D}_S B \to \mathsf{D}_{(R \sqcap S)}(A \times B)$$
$$\mathsf{n}^{R,S}_{A,B} : \mathsf{D}_{(R \sqcup S)}(A \times B) \to \mathsf{D}_R A \times \mathsf{D}_S B$$

natural in R, S, A, B. Thus the $\mathsf{m}^{R,S}_{A,X}$ operation combines an R-context and S-context into a single $(R \sqcap S)$-context, and $\mathsf{n}^{R,S}_{A,X}$ divides an $(R \sqcup S)$-context into R and S contexts. From a perspective of coeffects as shape annotations, $\mathsf{m}^{R,S}_{A,X}$ computes the intersection of the shapes and $\mathsf{n}^{R,S}_{A,X}$ divides the shape into (possibly overlapping) subshapes. These lax and colax operations provide the additional structure needed for the semantics.

Free-variable contexts. For every $A, B \in \mathbb{C}_{\mathsf{D}0}$ then $(A \times^D B) \in \mathbb{C}_{\mathsf{D}0}$ is the object $A \times B \in \mathbb{C}_0$. Given two $f : \mathsf{D}_X A \to B, g : \mathsf{D}_Y A \to C$ then a pairing can be defined using $\mathsf{n}^{R,S}_{A,B}$ to split the context as show in (51) in **Figure** 6.10. Projections are provided by π_1^{D} (52) and π_2^{D} (53) (**Figure** 6.10), which are restricted to products $\mathsf{D}_I(A \times B)$ due to the use of ε_I.

Function types. Exponents are necessarily indexed where, for all $A, B \in \mathbb{C}_{\mathsf{D}0}$ and all $X \in \mathbb{I}_0$, then $(A \Rightarrow^X B) \in \mathbb{C}_{\mathsf{D}0}$ is the object $(\mathsf{D}_X A \Rightarrow B) \in \mathbb{C}_0$.

$$\langle f, g\rangle^{\mathsf{D}} = \mathsf{D}_{X\sqcup Y}A \xrightarrow{\mathsf{D}\Delta} \mathsf{D}_{X\sqcup Y}(A\times A) \xrightarrow{\mathsf{n}^{X,Y}_{A,A}} \mathsf{D}_X A \times \mathsf{D}_Y A \xrightarrow{f\times g} B\times C \tag{51}$$

$$\pi_1^{\mathsf{D}} = \mathsf{D}_I(A\times B) \xrightarrow{\varepsilon_I} A\times B \xrightarrow{\pi_1} A \tag{52}$$

$$\pi_2^{\mathsf{D}} = \mathsf{D}_I(A\times B) \xrightarrow{\varepsilon_I} A\times B \xrightarrow{\pi_2} B \tag{53}$$

$$\Phi^{R,S}_{A,B} = \mathbb{C}(\mathsf{D}_{R\sqcap S}(A\times X), B) \xrightarrow{\mathbb{C}(\mathsf{m}^{R,S}_{A,X},-)} \mathbb{C}(\mathsf{D}_R A\times \mathsf{D}_S X, B) \xrightarrow{\phi_{A,B}} \mathbb{C}(\mathsf{D}_R A, \mathsf{D}_S X \Rightarrow B) \tag{54}$$

$$\Psi^{R,S}_{A,B} = \mathbb{C}(\mathsf{D}_R A, \mathsf{D}_S X \Rightarrow B) \xrightarrow{\psi_{A,B}} \mathbb{C}(\mathsf{D}_R A\times \mathsf{D}_S X, B) \xrightarrow{\mathbb{C}(\mathsf{n}^{R,S}_{A,X},-)} \mathbb{C}(\mathsf{D}_{R\sqcup S}(A\times X), B) \tag{55}$$

Figure 6.10. Additional operations for the semantics of the coeffect calculus

Currying, for abstraction. Currying between $\times^{\mathsf{D}}$ and $\Rightarrow^{\mathsf{D}}$ can be defined using $\mathsf{m}^{R,S}_{A,B}$, shown in equation (54), *i.e.*:

$$\Phi^{R,S}_{A,B}(f : \mathsf{D}_{R\sqcap S}(A\times X)\to B) = \phi_{A,B}(f\circ \mathsf{m}^{R,S}_{A,X}) : \mathsf{D}_R A \to (\mathsf{D}_S X \Rightarrow B)$$

Uncurrying, for application. Uncurrying can be defined using the $\mathsf{n}^{R,S}_{A,B}$ operation, shown in equation (55), *i.e.*:

$$\Psi^{R,S}_{A,B}(g : \mathsf{D}_R A \to (\mathsf{D}_S X \Rightarrow B)) = \psi_{A,B}\, g \circ \mathsf{n}^{R,S}_{A,X} : \mathsf{D}_{R\sqcup S}(A\times X)\to B$$

Figure 6.11 shows the full semantics, using the above operations and the usual scheme for the semantics of the simply-typed λ-calculus described in Chapter 2. Included is the semantics of [SUB], using the morphism mapping of D applied to the morphisms of $\mathbb{I}^{\mathsf{op}}$.

Remark 6.4.2. The coeffects for both [LET] and [APP] differ to those in the general coeffect system of **Figure** 6.6 (Appendix B.4.2 (p. 192) shows diagrams for [APP] and [LET] to illustrate the calculation of the coeffects). The differences are:

– [LET] - In the general coeffect system (**Figure** 6.6), the coeffect of [LET] is $R\sqcup(R\bullet S)$, whilst here in the semantics of **Figure** 6.11 the denotation expects a context indexed by coeffect $R\bullet(I\sqcup S)$. The two coeffects are however equivalent by the distributivity of $\bullet$ over $\sqcup$:

$$\begin{array}{lll} R\bullet(I\sqcup S) & = (R\bullet I)\sqcup(R\bullet S) & \{\bullet/\sqcup\ \textit{distributivity}\} \\ & = R\sqcup(R\bullet S) & \{(C,\bullet,I)\ \textit{monoid}\} \end{array} \tag{56}$$

Distributivity is part of the coeffect algebra definition and holds for all example calculi here.

– [APP] - In the general coeffect system, the coeffect of [APP] is $R\sqcup(S\bullet T)$, however here it is $(R\sqcup S)\bullet(I\sqcup T)$. These two are equal if $\bullet$ interchanges with $\sqcup$, *i.e.*,

$$\begin{array}{lll} R\sqcup(S\bullet T) = & (R\bullet I)\sqcup(S\bullet T) & \{(C,\bullet,I)\ \textit{monoid}\} \\ = & (R\sqcup S)\bullet(I\sqcup T) & \{\textit{interchange}\} \end{array} \tag{57}$$

This axiom holds only for resources, but not for liveness or dataflow, *e.g.*, for the dataflow calculus: $3\,\mathsf{max}\,(2+4) = 6 \neq (3\,\mathsf{max}\,2) + (0\,\mathsf{max}\,4) = 7$. However, if $\sqcup$ is idempotent then the coeffect for [APP] in the semantics here is always *greater* than the coeffect of [APP] in the

$$[\![\text{ABS}]\!]\frac{[\![\Gamma, x:\sigma\,?\,(R \sqcap S) \vdash e:\tau]\!] = k : \mathsf{D}_{R\sqcap S}(\Gamma \times \sigma) \to \tau}{\begin{array}{l}[\![\Gamma\,?\,R \vdash \lambda x.e : \sigma \xrightarrow{S} \tau]\!] = \Phi\, k : \mathsf{D}_R\Gamma \to (\mathsf{D}_S\sigma \Rightarrow \tau) \\ \qquad\qquad = \phi(k \circ \mathsf{m}^{R,S}_{\Gamma,\sigma}) : \mathsf{D}_R\Gamma \to (\mathsf{D}_S\sigma \Rightarrow \tau)\end{array}}$$

$$[\![\text{APP}]\!]\frac{[\![\Gamma\,?\,R \vdash e_1 : \sigma \xrightarrow{S} \tau]\!] = k_1 : \mathsf{D}_R\Gamma \to (\mathsf{D}_S\sigma \Rightarrow \tau) \qquad [\![\Gamma\,?\,T \vdash e_2 : \tau]\!] = k_2 : \mathsf{D}_T\Gamma \to \sigma}{\begin{array}{l}[\![\Gamma\,?\,(R \sqcup S) \bullet (I \sqcup T) \vdash e_1\, e_2 : \tau]\!] = \Psi k_1 \circ^{\mathsf{D}} \langle id^{\mathsf{D}}, k_2\rangle^{\mathsf{D}} : \mathsf{D}_{(R\sqcup S)\bullet(I\sqcup T)}\Gamma \to \tau \\ \qquad\qquad = \psi k_1 \circ \mathsf{n}^{R,S}_{\Gamma,\sigma} \circ ((\varepsilon_I \times k_2) \circ \mathsf{n}^{I,T}_{\Gamma,\Gamma} \circ \mathsf{D}\Delta)^{\dagger_{R\sqcup S, I\sqcup T}}\end{array}}$$

$$[\![\text{VAR}]\!]\frac{x_i : \tau_i \in \Gamma}{\begin{array}{l}[\![\Gamma \vdash x_i : \tau]\!] = \pi_i^{\mathsf{D}} : \mathsf{D}_I\Gamma \to \tau_i \\ \qquad\qquad = \pi_i \circ \varepsilon_I : \mathsf{D}_I\Gamma \to \tau_i\end{array}} \text{ where } |\Gamma| = n$$

$$[\![\text{LET}]\!]\frac{[\![\Gamma\,?\,S \vdash e_1 : \sigma]\!] = f : \mathsf{D}_S\Gamma \to \sigma \qquad [\![\Gamma, v:\sigma\,?\,R \vdash e_2 : \tau]\!] = g : \mathsf{D}_R(\Gamma \times \sigma) \to \tau}{\begin{array}{l}[\![\Gamma\,?\,R \bullet (I \sqcup S) \vdash \textbf{let } v = e_1 \textbf{ in } e_2]\!] = g \circ^{\mathsf{D}} \langle id^{\mathsf{D}}, f\rangle^{\mathsf{D}} : \mathsf{D}_{R\bullet(I\sqcup S)}\Gamma \to \tau \\ \qquad\qquad = g \circ ((\varepsilon_I \times f) \circ \mathsf{n}^{I,S}_{\Gamma,\Gamma} \circ \mathsf{D}\Delta)^{\dagger_{R,(I\sqcup S)}}\end{array}}$$

$$[\![\text{SUB}]\!]\frac{[\![\Gamma\,?\,S \vdash e : \tau]\!] = f : \mathsf{D}_S\Gamma \to \tau \qquad [\![S \sqsubseteq S']\!] = \iota^S_{S'} : S' \to S \in \mathbb{I}^{\mathsf{op}}{}_1}{[\![\Gamma\,?\,S' \vdash e : \tau]\!] = f \circ \mathsf{D}_{\iota^S_{S'}} : \mathsf{D}_{S'}\Gamma \to \tau}$$

Figure 6.11. Derived categorical semantics for the coeffect calculus.

general coeffect system of **Figure** 6.6, *i.e.*:

$$R \sqcup (S \bullet T) \sqsubseteq (R \sqcup S) \bullet (I \sqcup T) \tag{58}$$

which is proved as follows, where (58) implies $(R \sqcup (S \bullet T)) \sqcup ((R \sqcup S) \bullet (I \sqcup T)) = (R \sqcup S) \bullet (I \sqcup T)$:

$$\begin{array}{lll}
& (R \sqcup (S \bullet T)) \sqcup ((R \sqcup S) \bullet (I \sqcup T)) & \\
= & R \sqcup (S \bullet T) \sqcup (R \bullet I) \sqcup (R \bullet T) \sqcup (S \bullet I) \sqcup (S \bullet T) & \{\bullet/\sqcup \textit{ distributivity}\} \\
= & (R \bullet I) \sqcup (R \bullet T) \sqcup (S \bullet I) \sqcup (S \bullet T) & \{\textit{idempotent } \sqcup \textit{ and } \bullet/I \textit{ units}\} \\
= & (R \sqcup S) \bullet (I \sqcup T) & \{\bullet/\sqcup \textit{ distributivity}\} \\
\Rightarrow & R \sqcup (S \bullet T) \sqsubseteq (R \sqcup S) \bullet (I \sqcup T) \quad \square &
\end{array}$$

This holds for all of the example calculi here. Therefore, by the ordering of coeffects, the semantics of $[\![\text{APP}]\!]$ here describes a more general requirement than that specified by the coeffect system of the general coeffect calculus.

6.4.2. Refined semantics

The following refined semantics for application requires a context indexed by the coeffect for the [APP] rule in the general coeffect calculus (**Figure** 6.6), defined:

$$[\![\text{APP}]\!]\frac{[\![\Gamma\,?\,R \vdash e_1 : \sigma \xrightarrow{S} \tau]\!] = k_1 : \mathsf{D}_R\Gamma \to (\mathsf{D}_S\sigma \Rightarrow \tau) \qquad [\![\Gamma\,?\,T \vdash e_2 : \sigma]\!] = k_2 : \mathsf{D}_T\Gamma \to \sigma}{[\![\Gamma\,?\,R \sqcup (S \bullet T) \vdash e_1\, e_2 : \tau]\!] = \psi k_1 \circ (id \times k_2^{\dagger_{S,T}}) \circ \mathsf{n}^{R,S\bullet T}_{\Gamma,\Gamma} \circ \mathsf{D}\Delta : \mathsf{D}_{R\sqcup(S\bullet T)}\Gamma \to \tau}$$

which is illustrated by the following diagram:

$$\mathsf{D}_{R\sqcup(S\bullet T)}\Gamma \xrightarrow{\mathsf{D}\Delta} \mathsf{D}_{R\sqcup(S\bullet T)}(\Gamma\times\Gamma) \xrightarrow{\mathsf{n}^{R,S\bullet T}_{\Gamma,\Gamma}} \mathsf{D}_R\Gamma\times\mathsf{D}_{S\bullet T}\Gamma \xrightarrow{id\times(\mathsf{D}k_2\circ\delta_{S,T})} \mathsf{D}_R\Gamma\times\mathsf{D}_S\sigma \xrightarrow{\psi k_1} \mathsf{D}\tau \tag{59}$$

This semantics therefore refines the incoming contextual requirements to a more specialised (or more precise) coeffect. The refined semantics will be used from now on, unless otherwise stated. Section 6.4.3.1 shows the conditions under which the derived and refined semantics for application are equivalent.

6.4.3. Equational theory

Following the presentation in Chapter 2, and the comonadic semantics in Section 3.3.2, the usual $\beta\eta$-equalities are considered here for the coeffect calculus (using the refined semantics for application). This requires additional conditions on the indexed comonad and indexed (co)lax monoidal functor (with $\mathsf{m}^{R,S}_{A,B}$ and $\mathsf{n}^{R,S}_{A,B}$), analogous to those of co(lax) (semi)monoidal comonads (Section 3.3.2, p. 68).

Definition 6.4.3. An *indexed (co)lax semi-monoidal comonad* comprises an indexed comonad D with (co)lax semi-monoidal functor structure, *i.e.*, operations $\mathsf{m}^{R,S}_{A,B} : \mathsf{D}_R A\times\mathsf{D}_S B \to \mathsf{D}_{R\sqcap S}(A\times B)$ and $\mathsf{n}^{R,S}_{A,B} : \mathsf{D}_{R\sqcup S}(A\times B) \to \mathsf{D}_R A\times\mathsf{D}_S B$ (natural in R, S, A, B), where the counit and comultiplication operations of the indexed comonad commute with $\mathsf{m}^{R,S}_{A,B}$ and $\mathsf{n}^{R,S}_{A,B}$ (to reduce clutter, $\mathsf{m}^{R,S}_{A,B}$ and $\mathsf{n}^{R,S}_{A,B}$ are written $\mathsf{m}_{R,S}$ and $\mathsf{n}_{R,S}$, *i.e.*, the non-index objects are elided):

$$\begin{array}{ccccc}
\mathsf{D}_{X\bullet R}A\times\mathsf{D}_{Y\bullet S}B & \xrightarrow{\mathsf{m}_{X\bullet R,Y\bullet S}} & \mathsf{D}_{(X\bullet R)\sqcap(Y\bullet S)}(A\times B) & = & \mathsf{D}_{(X\sqcap Y)\bullet(R\sqcap S)}(A\times B) \\
\downarrow{\scriptstyle \delta_{X,R}\times\delta_{Y,S}} & & & & \downarrow{\scriptstyle \delta_{X\sqcap Y,R\sqcap S}} \\
\mathsf{D}_X\mathsf{D}_R A\times\mathsf{D}_Y\mathsf{D}_S B & \xrightarrow{\mathsf{m}_{X,Y}} & \mathsf{D}_{X\sqcap Y}(\mathsf{D}_R A\times\mathsf{D}_S B) & \xrightarrow{\mathsf{D}\mathsf{m}_{R,S}} & \mathsf{D}_{X\sqcap Y}\mathsf{D}_{R\sqcap S}(A\times B)
\end{array} \tag{60}$$

$$\begin{array}{ccc}
\mathsf{D}_I A\times\mathsf{D}_I B & \xrightarrow{\mathsf{m}_{I,I}} & \mathsf{D}_{I\sqcap I}(A\times B) \\
\downarrow{\scriptstyle \epsilon_I\times\epsilon_I} & & \| \\
A\times B & \xleftarrow[\varepsilon_I]{} & \mathsf{D}_I(A\times B)
\end{array} \tag{61}$$

$$\begin{array}{ccc}
\mathsf{D}_I A\times\mathsf{D}_I B & \xleftarrow{\mathsf{n}_{I,I}} & \mathsf{D}_{I\sqcup I}(A\times B) \\
\downarrow{\scriptstyle \epsilon_I\times\epsilon_I} & & \| \\
A\times B & \xleftarrow[\varepsilon_I]{} & \mathsf{D}_I(A\times B)
\end{array} \tag{62}$$

$$\begin{array}{ccccc}
\mathsf{D}_{R\bullet X}A\times\mathsf{D}_{S\bullet Y}B & \xleftarrow{\mathsf{n}_{R\bullet X,S\bullet Y}} & \mathsf{D}_{(R\bullet X)\sqcup(S\bullet Y)}(A\times B) & = & \mathsf{D}_{(R\sqcup S)\bullet(X\sqcup Y)}(A\times B) \\
\downarrow{\scriptstyle \delta_{R,X}\times\delta_{S,Y}} & & & & \downarrow{\scriptstyle \delta_{R\sqcup S,X\sqcup Y}} \\
\mathsf{D}_R\mathsf{D}_X A\times\mathsf{D}_S\mathsf{D}_Y B & \xleftarrow[\mathsf{n}_{R,S}]{} & \mathsf{D}_{R\sqcup S}(\mathsf{D}_X A\times\mathsf{D}_Y B) & \xleftarrow[\mathsf{D}\mathsf{n}_{X,Y}]{} & \mathsf{D}_{R\sqcup S}\mathsf{D}_{X\sqcup Y}(A\times B)
\end{array} \tag{63}$$

These axioms are analogous to the laws of a monoidal comonad, but now involve indices. The following additional coeffect axioms on $\sqcup$, $\sqcap$, and $\bullet$ are required for these diagrams to commute along the equality edges:

– $\delta_{R,S}$ is lax monoidal (60) when $\bullet$ and $\sqcap$ have an interchange law, *i.e.*: $(X\bullet R)\sqcap(Y\bullet S) = (X\sqcap Y)\bullet(R\sqcap S)$.

– ε_I is lax monoidal (61) when $I \sqcap I = I$.
– $\delta_{R,S}$ is colax monoidal (63) when $\bullet$ and $\sqcup$ have an interchange law, *i.e.*, $(X \bullet R) \sqcup (Y \bullet S) = (X \sqcup Y) \bullet (R \sqcup S)$.
– ε_I is colax monoidal (62) when $I \sqcup I = I$.

The following property is also used in this section for equational theories on the semantics:

Definition 6.4.4. An indexed (co)lax monoidal functor is *idempontent* if the following commute:

$$\begin{array}{ccc} \mathsf{D}_R A & \xrightarrow{\Delta} & \mathsf{D}_R A \times \mathsf{D}_R A \\ {\scriptstyle \mathsf{D}\Delta}\downarrow & & \downarrow{\scriptstyle \mathsf{m}^{R,R}_{A,A}} \\ \mathsf{D}_R(A \times A) & = & \mathsf{D}_{R \sqcap R}(A \times A) \end{array} \qquad \begin{array}{ccc} \mathsf{D}_R A \times \mathsf{D}_R A & \xleftarrow{\Delta} & \mathsf{D}_R A \\ {\scriptstyle \mathsf{n}^{R,R}_{A,A}}\uparrow & & \downarrow{\scriptstyle \mathsf{D}\Delta} \\ \mathsf{D}_{R \sqcup R}(A \times A) & = & \mathsf{D}_R(A \times A) \end{array} \tag{64}$$

Thus idempotency of a lax and colax monoidal functor implies the additional axioms of idempotency for $\sqcap$ and $\sqcup$ respectively. The example coeffect algebra all have these idempotency properties.

Remark 6.4.5 (Correspondence between identities on $\mathbb{I}$ and proofs)**.** There is a correspondence between algebraic identities on coeffects and coherence conditions between morphisms in $\mathbb{C}$. For example, the right-unit monoid axiom $X \bullet I = X$ corresponds to right-unit indexed comonad axiom $\mathsf{D}_X \varepsilon_I \circ \delta_{X,I} = id_{\mathsf{D}_X}$). This correspondence assists in proving equality of denotations, *e.g.*, for $[\![\Gamma \,?\, R \vdash e_1 : \tau]\!] \stackrel{?}{=} [\![\Gamma \,?\, S \vdash e_2 : \tau]\!]$ a proof that $R = S$ suggests the coherence conditions to prove equality of the denotations. This is made even more useful by the distinct algebraic operations on $\mathbb{I}$ used to calculate the coeffects of each indexed operation (indexed counit (I), indexed comultiplication ($\bullet$), indexed lax functor, ($\sqcap$), indexed colax functor ($\sqcup$)).

This correspondence is highlighted in this section for equational theories, where the semantic conditions are correlated with syntactic conditions.

It seems a stronger result likely holds if $f : \mathsf{D}_X A \to B$ and $g : \mathsf{D}_Y A \to B$ both uniquely factor through indexed operations which define X and Y, then a proof of $X \equiv Y$ maps directly to the proof that $f \equiv g$, modulo naturality properties and coherence conditions arising from non-indexed structures on $\mathbb{C}$ (such as products). Proving this result is further work.

6.4.3.1. Equality of the derived and refined semantics of application

Remark 6.4.2 (p. 128) showed that the refined and derived semantics for [APP] have equivalent coeffects if $\bullet$ and $\sqcup$ have an interchange law. This law corresponds to axiom that $\delta_{R,S}$ is colax monoidal (equation 63, p. 130), which requires this interchange law for axiom to hold. Following the correspondence between proofs on coeffect term equalities and coherence conditions on the semantics, the refined semantics for application is indeed equal to derived semantics for application when $\delta_{R,S}$ is a colax monoidal. The proof is provided in Appendix C.4.3 (p. 212).

6.4.3.2. $\beta\eta$-equality

Let-binding and abstract-application equality. As per the description in Chapter 2, [$\equiv$-*let*-λ] holds if $\Psi \circ \Phi = id$ (that is, currying is the right-inverse of uncurrying). For the contextual

λ-calculus in Section 3.3.2, this elicited the property that $\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id$. By analogy, a similar axiom is expected to hold for the indexed setting: $\mathsf{m}^{R,S}_{A,B} \circ \mathsf{n}^{R,S}_{A,B} = id$.

Two variants of [$\equiv$-*let*-λ] were suggested in Section 6.3.1.2 (p. 123):

1. *General*: $\Gamma \,?\, R \sqcup (S \bullet T) \vdash (\lambda v.e_1)\, e_2 \equiv \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau$.

   Syntactically, this equation requires idempotence of $\sqcup$, interchange between $\bullet$ and $\sqcup$, and for $\sqcap = \sqcup$. Proposition C.4.2 (p. 206) in the appendix shows the proof at the semantic level, which correspondingly requires idempotence of $\mathsf{n}_{R,S}$, colax monoidality of $\delta_{R,S}$, and $\mathsf{m}_{R,S} \circ \mathsf{n}_{R,S} = id$, inducing the condition that $\sqcap = \sqcup$.

2. *Restricted*: $\Gamma \,?\, R \sqcup (R \bullet S) \vdash (\lambda v.e_1)\, e_2 \equiv \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau$.

   Syntactically, this equation requires just idempotence of $\sqcap$, and distributivity of $\bullet$ and $\sqcup$ for *let*-binding. Proposition C.4.3 (p. 207) in the appendix shows the proof at the semantic level, which requires idempotence of $\sqcap$ but also idempotence of $\sqcup$ such that $\mathsf{m}^{R,R}_{A,B} \circ \mathsf{n}^{R,R}_{A,B} = id$ (note the indices are the same here), such that:

   $$\mathsf{D}_R A \times \mathsf{D}_R B \xrightarrow{\mathsf{m}^{R,R}_{A,B}} \mathsf{D}_{R \sqcap R}(A \times B) \equiv \mathsf{D}_R(A \times B) \equiv \mathsf{D}_{R \sqcup R} A \xrightarrow{\mathsf{n}^{R,R}_{A,B}} \mathsf{D}_R A \times \mathsf{D}_R B$$

   Furthermore, colax monoidality of $\delta_{R,S}$ is required, implying interchange of $\bullet$ and $\sqcup$ is needed. Syntactic equality of coeffects for [$\equiv$-*let*-λ] however requires only distributivity of $\bullet$ over $\sqcup$ and idempotence of $\sqcap$. The semantic and syntactic requirements for restricted [$\equiv$-*let*-λ] here do however correspond by the following result.

   Proposition 6.4.6. *Distributivity of $\bullet$ and $\sqcup$ is equivalent to interchange between $\bullet$ and $\sqcup$ and idempotency of $\sqcup$:*

   $$\begin{array}{lcll} R \bullet (S \sqcup T) & \equiv & (R \sqcup R) \bullet (S \sqcup T) & \{\sqcup \textit{ idempotence}\} \\ & \equiv & (R \bullet S) \sqcup (R \bullet T) \ \square & \{\bullet/\sqcup \textit{ interchange}\} \end{array}$$

   Therefore, a syntactic requirement for distributivity of $\bullet$ over $\sqcup$ corresponds to semantic requirements for a colax monoidal $\delta_{R,S}$ in a restricted setting (e.g., where $\delta_{X \sqcup Y, R \sqcup R}$ is used) and idempotence of $\mathsf{n}_{R,S}$ (such that $\sqcup$ is idempotent).

   Since the proof of restricted [$\equiv$-*let*-λ] (p. 207) uses colax monoidality of $\delta_{R,S}$ in the restricted case and idempotence of $\mathsf{n}_{R,S}$, these requirements correspond to the syntactic distributivity requirement.

β-equality. Given [$\equiv$-*let*-λ], [$\equiv$-β] is provided given an inductive proof for [*let*-β] showing that **let** is equivalent to substitution. Previously, Definition 6.3.3 (p. 124) showed the conditions of a *substituting coeffect algebra*, which provides additional axioms such that [*let*-β] holds syntactically (*i.e.*, the coeffects of substitution are equivalent to those of *let*). These additional axioms are that (1) I is either the least or greatest element of $\sqsubset$, (2) monoidal $(C, \sqcap, I)$, and (3) interchange between $\bullet$ and $\sqcap$.

Appendix C.4.3 (p. 207) provides the semantic proof of $[let\text{-}\beta]$, which requires the following semantic conditions:

- corresponding to (1) when I is the least, D has a colax monoidal functor with counit $\mathsf{n}_I : \mathsf{D}I \to I$ (meaning $I \sqcup R = R$) and $\mathsf{n}_{R,R}$ is idempotent).
- corresponding to (1) when I is the greatest, $\mathsf{n}_{I,I}$ is idempotent requiring $I \sqcup I = I$ which is implied by I being a limit (least or greatest).
- corresponding to (2), D is lax monoidal with unit $\mathsf{m}_I : I \to \mathsf{D}I$ meaning $I \sqcap R = R$.
- corresponding to (3), δ is lax (semi)monoidal.
- corresponding to distributivity of $\bullet$ over $\sqcup$, δ is colax monoidal in a restricted case (of the form $\delta_{X \sqcup Y, R \sqcup R}$) and $\mathsf{n}_{R,R}$ is idempotent (see Proposition 6.4.6 above).

η-equality. Similarly to the contextual λ-calculus in Chapter 3, $[\eta\text{-}\equiv]$ requires a colax monoidal comonad with the additional condition that, for the indexed (co)lax monoidal operations, $n^{R,S}_{A,B} \circ \mathsf{m}^{R,S}_{A,B} = id$, which therefore implies the requirement that $\sqcup = \sqcap$, with the composition:

$$\mathsf{D}_R A \times \mathsf{D}_S B \xrightarrow{\mathsf{m}^{R,S}_{A,B}} \mathsf{D}_{R \sqcap S}(A \times B) \equiv \mathsf{D}_{R \sqcup S}(A \times B) \xrightarrow{\mathsf{n}^{R,S}_{A,B}} \mathsf{D}_R A \times \mathsf{D}_S B$$

Proposition C.4.6 (p. 213) in the appendix shows the full proof for η-equality which elicits this requirements. The proof also shows that η-equality requires δ is colax (semi)monoidal in a restricted case (for $\delta_{R \sqcup S, I \sqcup I}$) and $\mathsf{n}_{I,I}$ is idempotent, corresponding to the distributivity axiom of coeffect algebra (see Proposition 6.4.6).

6.4.4. Example semantics

Example 6.4.7 (Liveness)**.** The liveness coeffect calculus (Section 6.2.1, p. 116) has a categorical semantics based on the *indexed partiality comonad* $\mathsf{P} : \mathbb{B}^{\mathsf{op}} \to [\mathbb{C}, \mathbb{C}]$ (Example 5.4.2, p. 105) with $\mathsf{P}_{\mathsf{F}} = 1$ and $\mathsf{P}_{\mathsf{T}} A = A$, $I = \mathsf{T}$ and $\bullet = \wedge$.

Note that previously P had $\mathbb{B}$ as its domain but here it has $\mathbb{B}^{\mathsf{op}}$. The opposite category is used such that the preorder, where $\mathsf{F} \sqsubseteq \mathsf{T}$, is modelled by the unique morphism $f : \mathsf{F} \to \mathsf{T} \in \mathbb{B}_1$ and thus $f^{\mathsf{op}} : \mathsf{T} \to \mathsf{F} \in \mathbb{B}^{\mathsf{op}}_1$ such that $\mathsf{P}_f : \mathsf{P}_{\mathsf{T}} \dot{\to} \mathsf{P}_{\mathsf{F}}$ is a natural transformation for the semantics of [SUB] defined $\mathsf{P}_f A = 1$; the opposite transformation $\mathsf{P}_{\mathsf{F}} \dot{\to} \mathsf{P}_{\mathsf{T}}$ is not definable.

The categorical semantics for liveness has the required additional indexed lax and colax monoidal structure, with $\sqcup = \vee$, and $\sqcap = \wedge$, defined:

$$\begin{array}{ll}
\mathsf{m}^{R,S}_{A,B} : \mathsf{P}_R A \times \mathsf{P}_S B \to \mathsf{P}_{R \sqcap S}(A \times B) & \mathsf{n}^{R,S}_{A,B} : \mathsf{P}_{R \sqcup S}(A \times B) \to \mathsf{P}_R A \times \mathsf{P}_S B \\
\mathsf{m}^{\mathsf{T},\mathsf{T}}_{A,B}(x, y) = (x, y) & \mathsf{n}^{\mathsf{T},\mathsf{T}}_{A,B}(x, y) = (x, y) \\
\mathsf{m}^{\mathsf{T},\mathsf{F}}_{A,B}(x, 1) = 1 & \mathsf{n}^{\mathsf{T},\mathsf{F}}_{A,B}(x, y) = (x, 1) \\
\mathsf{m}^{\mathsf{F},\mathsf{T}}_{A,B}(x, 1) = 1 & \mathsf{n}^{\mathsf{F},\mathsf{T}}_{A,B}(x, y) = (1, x) \\
\mathsf{m}^{\mathsf{F},\mathsf{T}}_{A,B}(x, 1) = 1 & \mathsf{n}^{\mathsf{F},\mathsf{F}}_{A,B} 1 = (1, 1)
\end{array}$$

As described in Section 6.3.1, liveness has β-equivalence and the restricted $[let\text{-}\equiv\text{-}\lambda]$ rule.

Example 6.4.8 (Dataflow)**.** The dataflow calculus (Section 6.2.2, p. 117) has coeffect algebra $(\mathbb{N}, +, 0, \mathsf{max}, \mathsf{min})$ and categorical semantics provided by the indexed stream comonad Stream :

$\mathbb{N}^{\mathsf{op}} \to [\mathbb{C}, \mathbb{C}]$, where $\mathsf{Stream}_N A = A^{(N+1)}$. The coeffect describes the number of elements in the past required by a computation, where there is always a current element (provided by the $+1$ on the exponent).

The operations of the indexed stream comonad are defined (for all $f : \mathsf{Stream}_R A \to B$):

$$f^{\dagger_{n,m}} : \mathsf{Stream}_{R+S}\, A \to \mathsf{Stream}_R\, B = \lambda a.\, \langle f\langle a_0, \ldots, a_S\rangle, \ldots, f\langle a_R, \ldots, a_{R+S}\rangle\rangle$$
$$\varepsilon_0 : \mathsf{Stream}_0\, A \to A = \lambda a. a_0$$

The lax and colax indexed (semi)-monoidal operations are defined:

$$\mathsf{n}^{X,Y}_{A,B} : \mathsf{Stream}_{R\mathsf{max}S}(A \times B) \to \mathsf{Stream}_R A \times \mathsf{Stream}_S B$$
$$\mathsf{n}^{X,Y}_{A,B} = \lambda\langle (a_0, b_0), \ldots (a_{X\mathsf{max}Y}, b_{X\mathsf{max}Y})\rangle.(\langle a_0, \ldots, a_X\rangle, \langle b_0, \ldots, b_Y\rangle)$$

$$\mathsf{m}^{X,Y}_{A,B} : \mathsf{Stream}_R A \times \mathsf{Stream}_S B \to \mathsf{Stream}_{R\mathsf{min}S}(A \times B)$$
$$\mathsf{m}^{X,Y}_{A,B} = \lambda(\langle a_0, \ldots, a_X\rangle, \langle b_0, \ldots, b_Y\rangle).\langle (a_0, b_0), \ldots, (a_{X\mathsf{min}Y}, b_{X\mathsf{min}Y})\rangle$$

The semantics of prev is given by $\mathsf{prev} : \mathsf{Stream}_1 A \to A = \lambda a. a_1$ and the additional rule:

$$\frac{[\![\Gamma ? N \vdash e : \tau]\!] = g : \mathsf{D}_N \Gamma \to \tau}{[\![\Gamma ? (N+1) \vdash \mathsf{prev}\, e : \tau]\!] = \mathsf{prev} \circ g^{\dagger_{1,N}} : \mathsf{D}_{N+1}\Gamma \to \tau}$$

As an example of the semantics in action, consider the term $(\lambda x.\mathsf{prev}\, x)(\mathsf{prev}\, y)$, which has type-and-coeffect derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, y : \tau, x : \tau \,?\, 0 \vdash x : \tau}[\textsc{var}]}{\Gamma, y : \tau, x : \tau \,?\, 1 \vdash \mathsf{prev}\, x : \tau}[\textsc{prev}]}{\Gamma, y : \tau \,?\, 1 \vdash \lambda x.\mathsf{prev}\, x : \tau \xrightarrow{1} \tau}[\textsc{abs}] \qquad \dfrac{\dfrac{}{\Gamma, y : \tau \,?\, 0 \vdash y : \tau}[\textsc{var}]}{\Gamma, y : \tau \,?\, 1 \vdash \mathsf{prev}\, y : \tau}[\textsc{prev}]}{\Gamma, y : \tau \,?\, (1 \,\mathsf{max}\, (1+1)) \vdash (\lambda x.\mathsf{prev}\, x)(\mathsf{prev}\, y) : \tau}[\textsc{app}]$$

i.e., the coeffect of the term is 2. The denotation is as follows (of type $\mathsf{D}_2(\Gamma \times \tau) \to \tau$):

$$\begin{array}{rl} & \psi(\phi(\mathsf{prev} \circ (\pi_2 \circ \varepsilon_0)^{\dagger_{1,0}} \circ \mathsf{m}_{1,1})) \circ (id \times ((\mathsf{prev} \ \circ (\pi_2 \circ \varepsilon_0))^{\dagger_{1,0}})^{\dagger_{1,1}}) \circ \mathsf{n}_{1,2} \circ \mathsf{D}\Delta \\ \equiv & \mathsf{prev} \circ (\pi_2 \circ \varepsilon_0)^{\dagger_{1,0}} \circ \mathsf{m}_{1,1} \circ (id \times ((\mathsf{prev} \ \circ (\pi_2 \circ \varepsilon_0))^{\dagger_{1,0}})^{\dagger_{1,1}}) \circ \mathsf{n}_{1,2} \circ \mathsf{D}\Delta \end{array}$$

where the second equation applies the adjunction law to simplify.

The following illustrates this semantics by showing intermediate stream objects between morphisms. An object $\mathsf{Stream}_N A$ is written $\langle a_0, \ldots, a_N\rangle$ with the following stream for the denotation of the top-level context:

$$\langle (a_0, b_0), (a_1, b_1), (a_2, b_2)\rangle : \mathsf{D}_2(\Gamma \times \tau)$$

For clarity, indexed coKleisli extensions $f^{\dagger_{R,S}}$ are expanded to $\mathsf{D}_R f \circ \delta_{R,S}$ (where subscripts on functors are elided for brevity), and $\mathsf{D}(g \circ f)$ is expanded to $\mathsf{D}g \circ \mathsf{D}f$.

$$
\begin{array}{lll}
 & \langle (a_0,b_0),(a_1,b_1),(a_2,b_2)\rangle & : \mathsf{D}_2(\Gamma \times \tau) \\
\to \mathsf{D}\Delta & = \langle ((a_0,b_0),(a_0,b_0)),((a_1,b_1),(a_1,b_1)),((a_2,b_2),(a_2,b_2))\rangle & : \mathsf{D}_2((\Gamma \times \tau) \times (\Gamma \times \tau)) \\
\to \mathsf{n}_{1,2} & = \langle (a_0,b_0),(a_1,b_1)\rangle \times \langle (a_0,b_0),(a_1,b_1),(a_2,b_2)\rangle & : \mathsf{D}_1(\Gamma \times \tau) \times \mathsf{D}_2(\Gamma \times \tau) \\
\to id \times \delta_{1,1} & = \langle (a_0,b_0),(a_1,b_1)\rangle \times \langle \langle (a_0,b_0),(a_1,b_1)\rangle, \langle (a_1,b_1),(a_2,b_2)\rangle\rangle & : \mathsf{D}_1(\Gamma \times \tau) \times \mathsf{D}_1\mathsf{D}_1(\Gamma \times \tau) \\
\to id \times \mathsf{D}\delta_{1,0} & = \langle (a_0,b_0),(a_1,b_1)\rangle \times \langle \langle \langle (a_0,b_0)\rangle, \langle (a_1,b_1)\rangle\rangle, \langle \langle (a_1,b_1)\rangle, \langle (a_2,b_2)\rangle\rangle\rangle & : \mathsf{D}_1(\Gamma \times \tau) \times \mathsf{D}_1\mathsf{D}_1\mathsf{D}_0(\Gamma \times \tau) \\
\to id \times \mathsf{DD}\varepsilon_0 & = \langle (a_0,b_0),(a_1,b_1)\rangle \times \langle \langle (a_0,b_0),(a_1,b_1)\rangle, \langle (a_1,b_1),(a_2,b_2)\rangle\rangle & : \mathsf{D}_1(\Gamma \times \tau) \times \mathsf{D}_1\mathsf{D}_1(\Gamma \times \tau) \\
\to id \times \mathsf{DD}\pi_2 & = \langle (a_0,b_0),(a_1,b_1)\rangle \times \langle \langle b_0,b_1\rangle, \langle b_1,b_2\rangle\rangle & : \mathsf{D}_1(\Gamma \times \tau) \times \mathsf{D}_1\mathsf{D}_1\tau \\
\to id \times \mathsf{Dprev} & = \langle (a_0,b_0),(a_1,b_1)\rangle \times \langle b_1,b_2\rangle & : \mathsf{D}_1(\Gamma \times \tau) \times \mathsf{D}_1\tau \\
\to \mathsf{m}_{1,1} & = \langle ((a_0,b_0),b1),((a_1,b_1),b2)\rangle & : \mathsf{D}_1((\Gamma \times \tau) \times \tau) \\
\to \delta_{1,0} & = \langle \langle ((a_0,b_0),b_1)\rangle, \langle ((a_1,b_1),b_2)\rangle\rangle & : \mathsf{D}_1\mathsf{D}_0((\Gamma \times \tau) \times \tau) \\
\to \mathsf{D}\varepsilon_0 & = \langle ((a_0,b_0),b_1),((a_1,b_1),b_2)\rangle & : \mathsf{D}_1((\Gamma \times \tau) \times \tau) \\
\to \mathsf{D}\pi_2 & = \langle b_1,b_2\rangle & : \mathsf{D}_1\tau \\
\to \mathsf{prev} & = b_2 & : \tau
\end{array}
$$

Thus, $(\lambda x.\mathsf{prev}\, x)(\mathsf{prev}\, y)$ computes the second to last value for y.

Example 6.4.9 (Resources)**.** The resource coeffect calculus (Section 6.2.3, p. 118) has a denotational semantics provided an *indexed product comonad* similar to that of Example 5.4.8 (p. 5.4.8) in the previous chapter. The resource calculus has coeffect algebra $(\mathcal{P}(\mathbb{R}), \cup, \emptyset, \cup, \cup)$ (where $\mathbb{R}$ is the set of pairs of resource identifiers (atoms) and their types) with indexed comonad $\mathsf{Res} : \mathcal{P}(\mathbb{R})^{\mathsf{op}} \to [\mathbb{C},\mathbb{C}]$ defined $\mathsf{Res}_X A = A \times X$. The indexed comonad operations are defined:

$$f^{\dagger_{n,m}} : \mathsf{Res}_{R \cup S}\, A \to \mathsf{Res}_R\, B = \lambda(a,x).(f(a, x - (S - R)), x - (R - S))$$

$$\varepsilon_0 : \mathsf{Res}_\emptyset\, A \to A = \lambda(a,\emptyset).a$$

where $x : R \cup S$ and thus the R-only subset of x is $x - (S - R)$ and the S-only subset of x is $x - (R - S)$ (where $(-)$ is set difference). The monoidal operations are defined:

$$\mathsf{n}^{R,S}_{A,B} : \mathsf{Res}_{R\cup S}(A \times B) \to \mathsf{Res}_R A \times \mathsf{Res}_S B = \lambda((a,b),x).((a, x - (S - R)),(b, x - (R - S)))$$

$$\mathsf{m}^{R,S}_{A,B} : \mathsf{Res}_R A \times \mathsf{Res}_S B \to \mathsf{Res}_{R\cup S}(A \times B) = \lambda((a,x),(b,y)).((a,b),(x \cup y))$$

This calculus has both β- and η-equality, since $(\mathcal{P}(\mathbb{R}), \cup, \emptyset, \cup, \cup)$ is a substituting coeffect algebra where $\sqcup = \sqcap$.

6.5. Haskell type constraints as coeffects

In Haskell, types have the general form $C \Rightarrow \tau$, comprising a type term τ and constraint term C. Such type constraints may include *type class* constraints (arising from the use of ad-hoc polymorphic expressions), *implicit parameter* constraints [**LLMS00**], and *equality* constraints [**SJCS08**]. Type constraints can be seen as a coeffect describing the contextual-requirements of an expression.[1] Therefore, ad-hoc polymorphic expressions can be seen as context-dependent computations, where the context provides the required definition of the expression at a particular type.

[1] Interestingly, for Haskell types $C \Rightarrow \tau$, the terms to left of $\Rightarrow$ are often called the *context* of the type [**HHPJW96**].

This section goes towards unifying coeffects and type constraints, which is used in the next chapter to encode a coeffect system based on sets using Haskell's type class constraints. Haskell's *implicit parameters* feature is considered first, which provides a calculus that resembles the *resources* language shown in Section 6.2.3.

6.5.1. Implicit parameters

The *implicit parameters* feature of Haskell, proposed by Lewis *et al.* [**LLMS00**], can be used to parameterise a computation without explicitly passing the parameters. This provides a solution to the "configuration problem" of converting a constant to a parameter in an existing program, which potentially requires large amounts of code rewriting to pass the parameter through the computation to the relevant subcomputation(s).

As Lewis *et al.* point out (in their *further work* section), comonads provide the mathematical structure of implicit parameters. They hypothesise that their translation is simply a rewriting of implicit parameters to the terms of a "family of coKleisli categories". This intuition is mostly correct. However, rather than a comonad, an *indexed comonad* structure is required to allow subcomputations with different implicit parameter requirements to be composed in a single program. The indexed product comonad can be used to provide a calculus for implicit parameters, where implicit parameter requirements are described by a coeffect system rather than having a uniform requirement for a whole program provided by the product comonad.

Implicit parameters in Haskell are denoted by a preceding question mark, and their use is tracked via special type constraints. For example, the following defines a function *format* with an implicit parameter $?width$ (implicit parameters are always prefixed by a question mark).

$$format :: (?width :: Int) \Rightarrow String \to String$$
$$format\ x = ...\mathbf{if}\ (length\ x) \geqslant ?width\ \mathbf{then}\ ...$$

The implicit parameter can be provided by a *let*-binding of $?width$, *e.g.*:

$$\mathbf{let}\ ?width = 80\ \mathbf{in}\ format :: String \to String$$

Lewis *et al.* present a type system for implicit parameter constraints where a special syntax replaces *let*-binding to provide a value to an implicit parameter. The type system extends standard Haskell typing with a set C of constraints for implicit parameters. Their approach can be expressed as a coeffect system with coeffect algebra $(\mathcal{P}(vars \times types), \cup, \emptyset, \cup, \cup)$, where *vars* denotes a set of syntactic variable names (atoms) (prefixed by a question mark) and *types* denotes the set of Haskell types (generated by Haskell's type constructors, constants, and universally quantified types). Thus the operations of the coeffect algebra are the same as that of the calculus of rebindable resources in Section 6.2.3. The coeffect approach describes the implicit parameter requirements of a function as latent coeffects, rather than constraints on the type in the approach of Lewis *et al.*.

Figure 6.12 shows the additional rules for introducing and discharging implicit parameter requirements. Requirements are discharged using the set difference operator $-$.

$$[?\textsc{let}]\frac{\Gamma \,?\, R \vdash e_1 : \tau_1 \qquad \Gamma \,?\, S \vdash e_2 : \tau_2}{\Gamma \,?\, ((R \cup S) \,-\, \{?x : \tau_1\}) \vdash \textbf{let}\ ?x = e_1\ \textbf{in}\ e_2 : \tau_2} \qquad [?\textsc{var}]\frac{}{\Gamma \,?\, \{?x : \tau\} \vdash ?x : \tau} \tag{65}$$

$$[?\textsc{let-fun}]\frac{\Gamma \,?\, R \vdash e_1 : \tau_1 \qquad \Gamma \,?\, S \vdash e_2 : \sigma \xrightarrow{T} \tau}{\Gamma \,?\, ((R \cup S) - \{?x : \tau_1\}) \vdash \textbf{let}\ ?x = e_1\ \textbf{in}\ e_2 : \sigma \xrightarrow{T - \{?x:\tau_1\}} (\tau - \{?x : \tau_1\})}$$

Figure 6.12. Additional coeffect rules for implicit parameters in Haskell

The [?LET-FUN] rule discharges requirements for an implicit parameter for a function. As an example, the following uses [?LET-FUN] and the non-deterministic coeffects in the usual [ABS] rule, where $\sqcap = \cup$ here, to associate the coeffects of a function to the outer context, where the requirement for the parameter $?x : \tau$ is then discharged by the binding of $?x$:

$$[?\textsc{let-fun}]\frac{\Gamma \,?\, R \vdash e : \tau \qquad [\textsc{abs}]\dfrac{[?\textsc{var}]\dfrac{}{\Gamma, v : \sigma \,?\, \{?x : \tau\} \vdash ?x : \tau}}{\Gamma \,?\, \{?x : \tau\} \vdash \lambda v.?x : \sigma \xrightarrow{\emptyset} \tau}}{\Gamma \,?\, R \vdash \textbf{let}\ ?x = e\ \textbf{in}\ \lambda v.?x : \sigma \xrightarrow{\emptyset} \tau} \tag{66}$$

Note, [?LET-FUN] discharges requirements recursively in the return type of the function, written as $(\tau - \{?x : \tau_1\})$ in **Figure** 6.12, *e.g.*, $\Gamma \,?\, \emptyset \vdash \textbf{let}\ ?x = e\ \textbf{in}\ (\lambda v.(\lambda y.?x)) : \sigma_1 \xrightarrow{\emptyset} (\sigma_2 \xrightarrow{\emptyset} \tau)$.

Figure 6.13 shows the type system for implicit parameters proposed by Lewis *et al.*, which has judgements of the form $C; \Gamma \vdash e : \tau$ meaning an expression e has type τ in context Γ with implicit parameter constraints C (a set of variable-type pairs). The main difference between this encoding and the coeffect encoding is that Lewis' approach has a single set of constraints C, with no latent constraints as in the coeffect calculus. This can be seen particularly in the (abs) rule of **Figure** 6.13, where the constraints of a function are those of the function body. This is analogous to how the [ABS] rule was used in the above example (equation (66)) associating the implicit parameter constraints of a function body with the immediate coeffects of the λ-abstraction with the empty set for the latent coeffects.

Related to this behaviour, (pvar) in **Figure** 6.13 gives the type of a polymorphic variable p which has a type class constraint D in its type in Γ. When p is used, its type variables may be instantiated with the assignment $[\bar{\alpha} \mapsto \bar{\tau}]$, and any constraints D included in the type of p in Γ are exposed as constraints of the term. In the coeffect calculus, these would be latent coeffects that would remain latent until the [APP] rule exposed them.

The (mvar) rule is similar to the [VAR] of the coeffect calculus, but instead of having an empty constraint, it has some arbitrary set of constraints C. Thus, (mvar) encodes constraint generalisation; (ivar) corresponds to the [?VAR] rule in **Figure** 6.12, but with implicit generalisation of constraints ([?VAR] records just the constraint of the implicit parameter term). The (with) rule is similar to [?LET], where, in the constraint of term e_1, $C \backslash ?x$ removes any $?x$ constraints from C, such that $C \backslash ?x, ?x : \tau$ means that there is precisely one constraint for $?x$ of type τ. This is then discharged by the **with** binding, which binds e_2 to $?x$ in the scope of e_1.

The two systems are equivalent in expressive power. Proving this formally is further work.

$$\text{(mvar)}\frac{x : \tau \in \Gamma}{C; \Gamma \vdash x : \tau} \quad \text{(ivar)}\frac{(?x : \tau) \in C}{C; \Gamma \vdash ?x : \tau} \quad \text{(pvar)}\frac{p : \forall \bar{\alpha}.D \Rightarrow \tau \in \Gamma \qquad D[\bar{\alpha} \mapsto \bar{\tau}] \subset C}{C; \Gamma \vdash p : \tau[\bar{\alpha} \mapsto \bar{\tau}]}$$

$$\text{(app)}\frac{C; \Gamma \vdash e_1 : \sigma \to \tau \qquad C; \Gamma \vdash e_2 : \sigma}{C; \Gamma \vdash e_1\, e_2 : \tau} \quad \text{(abs)}\frac{C; \Gamma, v : \sigma \vdash e : \tau}{C; \Gamma \vdash (\lambda v.e) : \sigma \to \tau}$$

$$\text{(with)}\frac{C \backslash ?x, ?x : \tau; \Gamma \vdash e_1 : \tau_1 \qquad C; \Gamma \vdash e_2 : \tau}{C; \Gamma \vdash (e_1 \,\mathbf{with}\, ?x = e_2) : \tau}$$

Figure 6.13. Implicit parameter type system of Lewis *et al.* [**LLMS00**].

Semantics of implicit parameters in Haskell. Implicit parameters in Haskell are translated into core Haskell code (a variation of System F) using the *dictionary passing* technique (also used for type class constraints [**WB89**]). In this technique, an expression of type $C \Rightarrow \tau$ becomes a function $C \to \tau$, where a class is represented as a dictionary of expression/functions for the class methods. Dictionary passing is captured by a kind of indexed product comonad, with $\mathsf{D}_C A = A \times \Pi_{?v:vars(C)} types(C, ?v)$, *i.e.*, a computation with implicit parameter requirements C is a pair of a value A and a dependent function mapping variables in C to values of their type.

6.5.2. Type class constraints

The use of ad-hoc polymorphic (*i.e.*, *overloaded*) expressions produces a contextual-requirement: that the context has an instance of the expression for a particular type. As mentioned above, Haskell desugars type class constraints into explicit dictionary parameters which provide the instance of the overloaded terms. A type class thus defines the polymorphic type of a dictionary of class methods, where a class instance provides a value of this dictionary type (possibly with some type instantiation). Thus, any expression that uses a class method incurs type class constraints which are desugared to dictionary parameters, whose types corresponds to the class of the constraint.

For example, the *Eq* class desugars to the specification of its dictionary, where its class methods are selectors on a dictionary, where **data** $EqDict\ a = EqDict\ (a \to a \to Bool)$:

$$(\equiv) :: EqDict\ a \to a \to a \to Bool$$
$$(\equiv)\ (EqDict\ eq)\ x\ y = eq\ x\ y$$

Thus, dictionaries are implicit parameters which can be captured by the indexed product comonad, *e.g.*, $(\equiv) :: \mathsf{D}_{EqDict\ a}\ a \to a \to Bool$.

In the same way as implicit parameter constraints, type class constraints are a coeffect system with a set-based coeffect algebra $(\mathcal{P}(\mathbb{C}), \cup, \emptyset, \cup, \cup)$ where $\mathbb{C}$ denotes the set of all possible type-class constraints, generated by the grammar:

$$C := () \mid (C_1, \ldots, C_n) \mid K\bar{\tau} \mid ?v : \tau$$

where $()$ is the empty constraint (denotes *true*), $(C_1, \ldots, C_n)$ is a conjunction of constraints, K denotes class constants (*e.g.*, *Eq*, *Ord*), and types are denoted τ, which may contain type

$$[\text{APP}]\frac{\Gamma \,?\, R \vdash e_1 : \sigma \xrightarrow{S} \tau \qquad \Gamma \,?\, T \vdash e_2 : \sigma}{\Gamma \,?\, R \cup S \cup T \vdash e_1\, e_2 : \tau} \cong [\text{APP}]\frac{\Gamma \vdash e_1 : (R, S) \Rightarrow \sigma \to \tau \qquad \Gamma \vdash e_2 : T \Rightarrow \sigma}{\Gamma \vdash e_1\, e_2 : (R, S, T) \Rightarrow \tau}$$

$$[\text{LET}]\frac{\Gamma, v : \tau' \,?\, R \vdash e : \tau \qquad \Gamma \,?\, S \vdash e' : \tau'}{\Gamma \,?\, R \cup S \vdash \textbf{let}\ e' = v\ \textbf{in}\ e : \tau} \cong [\text{LET}]\frac{\Gamma, v : \tau' \vdash e : R \Rightarrow \tau \qquad \Gamma \vdash e' : S \Rightarrow \tau'}{\Gamma \vdash \textbf{let}\ e' = v\ \textbf{in}\ e : (R, S) \Rightarrow \tau}$$

$$[\phi]\frac{}{\Gamma \,?\, R \vdash \phi : \tau} \cong [\phi]\frac{}{\Gamma \vdash \phi : R \Rightarrow \tau}$$

Figure 6.14. Encoding of set-based coeffects for a first-order calculus as Haskell type-class constraints

variables. Constraints in Haskell can include implicit parameter constraints (see the previous section) and type equality constraints $\tau_1 \sim \tau_2$ (not discussed here).

Type class instances provide the definitions that satisfy the type class constraints (and therefore discharge type-class constraints). The rules for discharging constraints are not considered here since they are complex, related to unification in the polymorphic λ-calculus.

In the next chapter, type class constraints are used to encode a set-based coeffect system in Haskell, for a first-order language without λ-abstraction. In this encoding, the coeffect of an expression $\Gamma \,?\, R \vdash e : \tau$ is represented as a constraint $\Gamma \vdash e : R \Rightarrow \tau$ using the scheme of mappings shown in **Figure** 6.14 between coeffects and Haskell typing rules involving type class constraints. The $[\phi]$ rule captures a constant for an application-specific operation which is given the type-class constraint (coeffect) R.

The typing rules on the right are valid in Haskell, representing set-based coeffects. This representation, or encoding, is useful since the idempotency, associativity, units, and symmetric properties of a set-based coeffect algebra are provided for free by Haskell's constraint checker, *e.g.*, $(Eq\ \ a, Ord\ \ a) \equiv (Ord\ \ a, Eq\ \ a) \equiv (Eq\ \ a, (Eq\ \ a, Ord\ \ a))$. A type-based encoding would need to manually handle these properties, requiring considerable effort.

6.6. Conclusion

6.6.1. Related work

Several existing type systems can be described as coeffect systems. The implicit parameters of Haskell for example was shown to have an equivalent coeffect system in Section 6.5.1, related to the calculus of dynamically-bound resources in Section 6.2.3. The resources example was also considered in the context of distributed environments. The literature includes various approaches to distributed programming, often with inference rules tracking the location in which a computation runs, or may run, *e.g.* [**MCH08**, **SC10**, **CLWY06**].

Type systems which guarantee secure information flow (*e.g.* [**SM03**, **VIS96**]) may also be viewed as coeffect systems, where coeffects specify the secrecy of the data being computed (or input). Related to security applications, the *calculus of capabilities* by Crary *et al.* describes contexts which carry *capabilities* that allow access to memory regions [**CWM99**]. Future work is

to apply the general coeffect calculus described in this dissertation to these distributed, security, and capability settings, to see if it provides sufficient generality capture these existing systems.

6.6.2. Further work

Structural coeffects. A limitation of the general coeffect system described in this chapter is that it approximates coeffects over the whole context of free variables. This over-approximation is significant for some of the coeffect systems described. For example, the liveness analysis tracks whether any variable is live, rather than specific variables.

Further work is to extend the system to track coeffects *per variable* with *structural coeffects*. One approach could be to replace the $\sqcap$ and $\sqcup$ operations for splitting and joining coeffects with an injective operation $\times$. The rules for abstraction and application then become:

$$[\text{ABS}]\frac{\Gamma, v : \sigma \,?\, R \times S \vdash e : \tau}{\Gamma \,?R \vdash \lambda v.e : \sigma \xrightarrow{S} \tau} \quad [\text{APP}]\frac{\Gamma \,?\, S \vdash e_1 : \sigma \xrightarrow{T} \tau \qquad \Gamma' \,?\, R \vdash e_2 : \sigma}{\Gamma, \Gamma' \,?\, S \times (T \bullet R) \vdash e_1\ e_2 : \tau} \quad [\text{VAR}]\frac{}{x : \tau \,?\, I \vdash x : \tau}$$

This system would then require explicit structural rules that manipulate the coeffects accordingly to the changes in the free-variable context, for example:

$$[\text{WEAKEN}]\frac{\Gamma \,?\, R \vdash e : \tau}{\Gamma, x : \sigma \,?\, R \times I \vdash e : \tau} \quad [\text{EXCH}]\frac{\Gamma, \Gamma' \,?\, R \times S \vdash e : \tau}{\Gamma', \Gamma \,?\, S \times R \vdash e : \tau}$$

A generalisation of the coeffect calculus described in this dissertation may capture both the original formulation and this structural system, providing a system that can be specialised to this more precise system or the original approximate system.

Optimisations. Coeffects can provide an optimisation via two-calling conventions for contextual computations: *call-by-intension* and *call-by-extension*. For computations which have I as a coeffect, it is likely that their denotation factors through the counit operation of the underlying indexed comonad, *i.e.*, given $[\![\Gamma \,?\, I \vdash e : \tau]\!] : \mathsf{D}_I\Gamma \to \tau$ then there exists an $f' : \Gamma \to \tau$ such that $f = f' \circ \varepsilon_I$. In this case an optimisation can be provided by rewriting the denotation of a computation to use the *call-by-extension* convention, where an extensional value of type Γ is passed to the denotation of e (provided instead by f'), using the composition of the underlying category rather than the indexed coKleisli composition. This convention is contrasted with the usual semantics which is *call-by-intension*. Further work is to explore under what conditions any morphism $\mathsf{D}_I\Gamma \to \tau$ can be shown to factor through ε_I.

This optimisation is in some sense dual to the *strictness* optimisation for effects [**Myc80**], where a parameter is passed as call-by-value rather than call-by-need for parameters that are definitely evaluated (strict). Further work is to further develop this optimisation and call-by-intension/extension.

6.6.3. Conclusion

This chapter introduced the general notion of a *coeffect* calculus for contextual computations, which comes with a *coeffect system* describing a static analysis which reconstructs the contextual properties of a program. A categorical semantics was provided by *indexed comonads*, with additional indexed monoidal structure. The propagation of coeffect information in a coeffect system corresponds exactly to the semantics propagation of context in the categorical semantics, as witnessed by the indices of the indexed comonad structure which provides the coeffect information. Promising further work is to generalise the coeffect calculus further to encapsulate other existing systems and to include *structural* notions of coeffects.

The simple dataflow calculus of Section 6.2.2 (p. 117) described the number of elements accessed into the past using prev via its coeffect system. Equivalently, the coeffect describes the data access pattern of an expression on a stream (following from its semantics using streams to model the context). This coeffect may be generalised to any (indexed) comonad that has a finite set of data accessors. For example, a coeffect system might analyse data access patterns on arrays in an array-programming language with the goal of statically excluding out-of-bounds access. Thus, relative data access is a contextual property (coeffect) of local array operations (also known as *stencils*). The data-access pattern of a stencil may be tracked by a set of the relative indices accessed from the arrays in the context:

$$\frac{\Gamma \,?\, F \vdash e : \mathsf{Array}\,\tau \qquad \Gamma \,?\, \emptyset \vdash i : \mathbb{Z}^n}{\Gamma \,?\, F \cup \{i\} \vdash e[i] : \tau}$$

for the access of a relative index n, which has no coeffects itself, from e. A local-mean operation on a one-dimensional array therefore has coeffect/type judgement:

$$x : \mathbb{R} \,?\, \{-1, 0, 1\} \vdash (x[-1] + x[1] + x[0]) \,/\, 3.0 : \mathbb{R}$$

The semantics of this stencil language might be provided by the *pointed array comonad* (Chapter 3), where the coeffects of a stencil give a requirement on the amount of *boundary* elements that are required for the array such that the relative access is also defined (*i.e.* does not cause an out-of-bounds error). Thus, the local mean operations requires a one-dimensional array x with one element of *exterior* elements defined in each direction.

This kind of *data access* analysis forms the basis of the next chapter, which describes a contextual language for programming with containers. An indexed array comonad is used, where the coeffects/indices describe the contextual requirement of a stencil's data access pattern. A coeffect analysis on data access patterns provides static safety invariants. This contextual language is embedded in Haskell and using the above approach of Section 6.5.2 for encoding coeffects using Haskell's type class constraints, coupled with a notation akin to **codo**.

Chapter 7

A contextual language for containers

This chapter unites the work of the previous chapters into a single programming language, called *Ypnos*, for programming with containers which permits efficient and parallel executions. Ypnos uses the notation of Chapter 4, extending this with a novel, specialised pattern matching notation. Ypnos is parameterised in its back-end by different containers that are *(relative) indexed comonads* (from Chapter 5). Furthermore, Ypnos has an associated coeffect system, following the work of Chapter 6.

This chapter is based on the papers *Ypnos: declarative, parallel structure grid programming* [**OBM10**], in collaboration with Max Bolingbroke and Alan Mycroft, and *Efficient and Correct Stencil Computation via Pattern Matching and Static Typing* with Alan Mycroft [**OM11**]

Introduction. One of the motivating examples of contextual computations in the introductory chapter was of data transformations described by local operations on a structure. This pattern is pervasive and also highly data-parallel, known as the *structured grid* (for arrays) and *unstructured grid* (for graphs) computational patterns [**ABC$^+$06, ABD$^+$08**]. For arrays, the local operations of these comonads are also commonly known as *stencil computations*.

Given the current trends towards parallel hardware, in response to diminishing returns from traditional architectures, language support for parallelism is crucial to making effective use of hardware. The design ethos behind Ypnos is provided by the rubric of the *four Rs* (see Chapter 1): to improve the readability and writability of programs, to improve their efficiency of their execution and facilitate parallel executions, and to provide strong program invariants to aid reasoning. To achieve these goals, Ypnos is designed as a *domain specific language*, that is *embedded*, or *internal*, to Haskell. In the restricted setting of a domain-specific language, the four Rs are more easily achieved without significant trade-offs, simultaneously providing problem specific expressivity (improving programmer productivity), stronger reasoning, and more appropriate optimisations than general-purpose languages [**Ste09, Orc11**].

As an embedded language, Ypnos provides a mix between contextual categorical programming, and programming in a contextual language. Ypnos is parameterised by a relative indexed comonad (Section 5.5.2) but currently provides the greatest support for programming with arrays (which are called *grids* to avoid the implementational connotations of arrays) with specialised syntactic extensions.

Chapter structure. Section 7.1 introduces the Ypnos language, including its coeffect system. Example 7.1.1 (p. 150) summarises the language features. Section 7.2 describes the underlying mathematical structure of Ypnos and its implementation. Section 7.3 giving some performance measures for a prototype of Ypnos.

Most of this chapter focuses on the array support in Ypnos. Section 7.4 discusses other containers and describes container comonads in terms of their *navigation operations*, defining *adjacent* contexts to the current. Section 7.5 concludes with related and further work.

7.1. Introducing Ypnos

Ypnos comprises a number of built-in data types, library functions, and syntactic extensions. The syntax of Ypnos is mostly that of Haskell, with extensions including a form of the **codo**-notation of Section 4.1, a specialised pattern matching syntax, called *grid patterns*, and a syntax for defining the *boundary behaviour* of a grid (defining values outside a grid's domain of indices).

Ypnos-specific syntax is written inside of *quasiquoting* brackets [**Mai07**] and expanded by macro functions, of the form:

[**dimensions** | *...ypnos code...* |]
[**stencil** | *...ypnos code...* |]
[**boundary** | *...ypnos code...* |]

The macros essentially give an executable semantics to the specialised syntax of Ypnos.

7.1.1. Grids and dimensions

Grids in Ypnos abstract over multi-dimensional arrays, where the Grid data type has three parameters: Grid d b a, where d describes the dimensionality of the grid, b records information about the *boundary* of a grid, and a is the element type.

Type-level dimension information comprises tensor products of dimension names, where the names of dimensions used in a program must be first declared. For example, the following declares dimensions named X and Y:

[**dimensions** | X, Y |]

At the type-level dimension names appear as parameters to the Dim constructor. The operator $:*$ takes the tensor product of dimensions. For example, the type of two-dimensional grids with dimensions X and Y is Grid (Dim X $:*$ Dim Y).

The grid and listGrid operations construct grids from a list of index-value pairs and a list of values respectively. Each takes five parameters: a dimension term, the least index of the grid (lower extent), the greatest index of the grid (upper extent), the list of the grid's data, and structure which describes the *boundary behaviour* of the grid (explained shortly).

7.1.2. Stencils

The **stencil** macro is a variation of the **codo**-notation of Chapter 4, where the **codo** keyword is elided. This *stencil notation* requires that the container data type to which it is applied is a relative comonad. Indeed, the Grid data type has a *relative comonad* structure.

The stencil notation therefore assists in constructing stencil computations from a number of other local operations on containers, *e.g.*

$$\begin{aligned} \mathit{gaussDiff2D} = [\mathbf{stencil} \mid x \Rightarrow\ & y \leftarrow \mathit{gauss2D}\ x \\ & z \leftarrow \mathit{gauss2D}\ y \\ & (\mathsf{current}\ y) - (\mathsf{current}\ z)\ |] \end{aligned}$$

Stencil functions can be applied using the extension of the comonad, which in Ypnos is called `apply`. In the usual way, `apply` applies a stencil function at each position in the parameter grid by instantiating the internal cursor of the grid to each index, writing the results into a new grid.

The stencil notation however differs from the **codo**-notation in that it also provides a special pattern matching syntax called *grid patterns*.

7.1.3. Grid patterns and coeffects

Ypnos has a coeffect system tracking the data access pattern of local operations, which is used to enforce safety guarantees and inform optimisations. As suggested informally in Section 6.6.3 (p. 141), data access on an n-dimensional array can be tracked as a coeffect, with the following rule for indexing:

$$\frac{\Gamma \,?\, F \vdash e : \mathsf{Array}\,\tau \qquad \Gamma \,?\, \emptyset \vdash i : \mathbb{Z}^n}{\Gamma \,?\, F \cup \{i\} \vdash e[i] : \tau}$$

Arrays may be indexed by general expressions, thus a more appropriate rule might be:

$$\frac{\Gamma \,?\, F \vdash e : \mathsf{Array}\,\tau \qquad \Gamma \,?\, \emptyset \vdash e' : \mathbb{Z}^n \qquad e' \rightsquigarrow i}{\Gamma \,?\, F \cup \{i\} \vdash e[e'] : \tau}$$

where $e' \rightsquigarrow i$ means that e' is statically reducible to a ground term i. This implies that the coeffect is in general undecidable. Rather than providing an approximate analysis, that in some cases may fail, Ypnos replaces general indexing with a form of relative indexing for which the coeffect analysis is decidable. Relative indexing is provided via a novel pattern matching syntax called *grid patterns* which binds values of a grid relative to the cursor.

The following is an example of one-dimensional grid pattern for a grid of dimension X:

$$X : |\ l\ \ @c\ r\ |$$

which is equivalent to general indexing $l = A[i-1]$, $c = A[i]$, and $r = A[i+1]$ for an array A and cursor i. The @ symbol marks the pattern for the cursor element. The relative lexical position of the remaining patterns to the cursor pattern determines the element in the grid, relative to the cursor, to which each pattern should be matched against.

The subpatterns of a grid pattern are restricted to either variable patterns, wildcard patterns, or further grid patterns. Therefore, values cannot be matched against. Grid patterns can be nested to provide n-dimensional grid patterns. For example, a pattern match on a two-dimensional grid, of dimensionality $\mathsf{Dim}\ X :* \mathsf{Dim}\ Y$, may be written:

$$\begin{aligned} Y : |\quad X : |\ nw\ \ @n\ \ ne\ | \\ @X : |\ w\ \ @c\ \ e\ | \\ X : |\ sw\ \ @s\ \ se\ || \end{aligned}$$

The dimension identifiers therefore disambiguate the dimension being matched upon by the pattern, where the outer pattern matches in the Y dimension and inner patterns in the X dimension. The application of a one-dimensional grid pattern to an n-dimensional (where $n > 1$) grid therefore creates array slices in the remaining dimensions. Note that every grid pattern must contain exactly one subpattern prefixed by @, where a nested grid pattern must have its own cursor marker @.

For syntactic convenience, Ypnos provides a two-dimensional grid pattern syntax which is easier to read and write than nested patterns. The above pattern can be written as the following two-dimensional grid pattern:

$$\begin{array}{lllll} X :* \ Y : & | \ nw & n & ne & | \\ & | \ w & @c & e & | \\ & | \ sw & s & se & | \end{array}$$

Grid patterns can be used in the stencil notation wherever a pattern may appear, which have only a single case (*i.e.*, the pattern is *irrefutable*). For example, the two-dimensional Gaussian operator can be defined:

$$\begin{array}{lllll} gauss2D = [\mathbf{stencil} \mid X :* \ Y : & | \ \_ & n & \_ & | \\ & | \ w & @c & e & | \\ & | \ \_ & s & \_ & | \ \Rightarrow (n + w + e + s + 2 * c) \ / \ 6.0 \ |] \end{array}$$

There is no other indexing mechanism for grids therefore grid patterns restrict the programmer to relative indexing, expressed as a static construction that requires no evaluation or reduction. Furthermore, grid patterns cannot contain value patterns, therefore, well-typed grid patterns cannot fail to match and are thus total. Grid patterns therefore provide an indexing scheme with a decidable coeffect analysis of the access pattern, providing decidable compile-time information, without the need for an approximate analysis.

Coeffects. The stencil notation in Ypnos has a coeffect analysis that tracks the data access pattern of the stencil. Chapter 6 described the general coeffect system parameterised by a coeffect algebra $(C, \bullet, I, \sqcup, \sqcap)$. Since the stencil notation (**codo**-notation) provides only *let*-binding and not λ-abstraction, the $\sqcap$ operation of a coeffect algebra is not required (since it is used exclusively for calculating the coeffects of λ-abstraction). Instead only a subset of the regular coeffect algebra structure is needed, with $(C, \bullet, I, \sqcup)$. This can be described as a *first-order coeffect algebra* since the operation for handling the coeffects of abstraction is removed. Coeffects in Ypnos are sets of relative indices, with the coeffect algebra $(\mathcal{P}((\mathbb{Z}^n), \cup, \emptyset, \cup)$ for data access on an n-dimensional array.

Figure 7.1 shows the coeffects for the stencil notation; [STENCIL-BIND] augments the usual typing rule for binding in the **codo**-notation with coeffects and is akin to the [LET] rule for set-based coeffects (*e.g.*, coeffects of resources in Section 6.2.3, p. 118) with standard pattern matches (not grid patterns). The [STENCIL-GRIDP] rule describes the coeffects of a grid pattern, where the lexical position of a variable pattern relative to the cursor pattern determines the

$$[\textsc{stencil-bind}]\ \frac{\Gamma;\Delta\,?\,R \vdash_c e : \tau \qquad \Gamma;\Delta,\Delta'\,?\,S \vdash_c r : \tau' \qquad dom(\Delta') = FV(p)}{\Gamma;\Delta\,?\,R \cup S \vdash_c p \leftarrow e;\ r : \tau'}$$

$$[\textsc{stencil-gridp}]\ \frac{R = \{i | vars(gp_i)\} \qquad \Gamma;\Delta\,?\,S : \sigma \vdash_c e : \tau \qquad dom(\Delta) = FV(gp)}{\Gamma \vdash [\,\mathbf{stencil} \mid d\text{:} \mid gp \mid \Rightarrow e \mid] : \mathsf{Grid}\ d\ b\ \sigma \xrightarrow{R \cup S} \tau}$$

$$[\textsc{stencil}]\ \frac{\Gamma;\Delta\,?\,S : \sigma \vdash_c e : \tau \qquad dom(\Delta) = FV(p)}{\Gamma \vdash [\,\mathbf{stencil} \mid d : p \Rightarrow e \mid] : \mathsf{Grid}\ d\ b\ \sigma \xrightarrow{S} \tau}$$

Figure 7.1. Coeffects for Ypnos stencil notation

coeffect, *e.g.*, for one-dimensional patterns: $|p_{-m} \ldots @p_0 \ldots p_n|$ the coeffect includes the relative index i if p_i is a variable pattern (the subscripts are the lexical position of the variable relative to the cursor pattern). The [stencil-gridp] rule thus computes the set R of the relative indices for the patterns within a grid pattern that are variable patterns (tested by the *vars* predicate). [stencil] types stencils that do not have a grid pattern on their parameter.

For example, the grid pattern for the *gauss2D* stencil function has the following coeffects:

$$\frac{\Gamma; n : \sigma, e : \sigma, w : \sigma, s : \sigma \vdash_c \ldots : \tau}{[\,\mathbf{stencil} \mid X :* Y : \left|\begin{array}{ccc} \_ & n & \_ \\ w & @c & e \\ \_ & s & \_ \end{array}\right| \Rightarrow \ldots \mid] : \mathsf{Grid}\ (\mathsf{Dim}\ X :* \mathsf{Dim}\ Y)\ b\ \sigma \xrightarrow{\{(-1,0),(0,-1),(1,0),(0,1)\}} \tau}$$

The overall coeffect of a term in the stencil notation becomes a latent coeffect of the defined function (see [stencil-gridp] and [stencil]). This latent coeffect describes the (relative) data access of the stencil and is used to enforce a safety property: that the parameter grid to the stencil has adequate *boundary values* such that the stencil can be applied at all indices. Boundary values are required for those indices that are outside the domain of the array; *e.g.*, for *gauss2D*, at the cursor $(0, 0)$, the indices $(-1, 0)$ and $(0, -1)$ are accessed, which if unchecked would cause an *out-of-bounds* error.

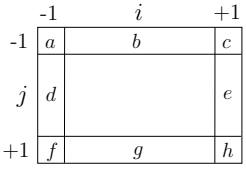
Section 7.2 describes how this latent coeffect is encoded using Haskell type-class constraints.

7.1.4. Boundaries

Section 3.1.1 (p. 50) defined the *gauss2D* and *laplace2D* local operations for the pointed array comonad using an indexing operation (?) which included boundary checking and default boundary values for indices outside the *extent* of the array.

Bounds-checking each array access is however inefficient, especially since the majority of accesses are to indices *inside* a grid's extent. A common solution is to expand the size of an array to include boundary values as array elements, called *exterior elements*. A traversal of the array is then made over the *interior elements*, that is, the non-boundary data for the array. This associates the boundary behaviour of a stencil operations to the array data structure.

Ypnos follows this approach, where grids are constructed with both exterior and interior elements. To apply a stencil function to a grid, the grid must have sufficient exterior elements



$$\begin{array}{llll} a = & (-1,-1) & e = & (+1,j) \\ b = & (i,-1) & f = & (-1,+1) \\ c = & (+1,-1) & g = & (i,+1) \\ d = & (-1,j) & h = & (+1,+1) \end{array}$$

Figure 7.2. Boundary region descriptors for two-dimensional grids with one-element exterior.

such that the stencil function does not access elements outside the grid's domain. Any program without appropriate boundaries for a stencil is rejected at compile-time.

Ypnos provides additional syntax for defining the boundary behaviour, via the **boundary** quasiquoted macro. The syntax defines a boundary structure which is used as a parameter to a grid constructor. Boundary terms specify the boundary behaviour in terms of *boundary regions* outside of a grid's domain. **Figure** 7.2 illustrates the boundary regions to a depth of one element in each direction of a two-dimensional array.

Boundary terms have the following syntax (where n are integers greater than one, e are Haskell expressions, and v are variables):

$$\begin{aligned} E &::= D : \overline{B} \\ B &::= \textbf{from}\ R_1\ \textbf{to}\ R_2 \to e \mid R \to e \mid R\ v \to e \\ R &::= P \mid (P,P) \mid (P,P,P) \mid (P,P,P,P) \mid \ldots \\ P &::= \texttt{-}n \mid \texttt{+}n \mid v \end{aligned}$$

Thus, a boundary definition comprises a dimension term D and a number of B terms specifying the value for a range of boundary regions, or the value of a specific boundary region, where R are *region descriptors*. The three forms of region definitions are called the *range form*, the *specific form*, and the *parameterised specific form*. The terminal $+n$ denotes a boundary region n elements after the upper extent of a grid in some dimension; $-n$ denotes a boundary region n elements before the lower extent of a grid in some dimension; v denotes any index inside the extent of a grid in some dimension where the value of the index is bound to v (see **Figure** 7.2).

There are various types of boundary condition that are well-known in the field of numerical analysis, used for different purposes; three common boundary conditions and their programming in Ypnos are described here, demonstrating the boundary syntax.

***Dirichlet* boundary condition.** *Dirichlet* conditions provide constant values outside the bounds of a problem. The *range form* and *specific form* of Ypnos' boundary syntax provides Dirichlet boundaries with constant values. For example, a Dirichlet boundary of value 0.0 is defined:

$$[\textbf{boundary} \mid X :* Y : \textbf{from}\ (-1,-1)\ \textbf{to}\ (+1,+1) \to 0.0 \mid]$$

The **from** ... **to**... syntax is itself syntactic sugar for a more verbose description specifying sub-regions of the exterior elements. The above is short-hand for the following cases:

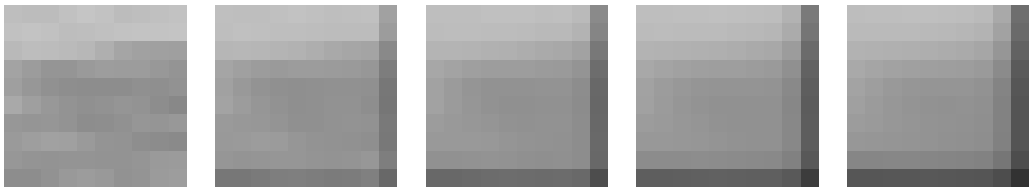
$$
\begin{aligned}
[\textbf{boundary} \mid X :* Y : (-1,-1) &\to 0.0 \\
(i,-1) &\to 0.0 \\
(+1,-1) &\to 0.0 \\
(-1,j) &\to 0.0 \\
(+1,j) &\to 0.0 \\
(-1,+1) &\to 0.0 \\
(i,+1) &\to 0.0 \\
(+1,+1) &\to 0.0 \mid]
\end{aligned}
$$

One disadvantage of the Dirichlet condition is that it introduces spurious data, called *artefacts*, where the constant boundary values *leak* into an image, as illustrated in **Figure** 7.3. For some programs this behaviour is useful, for example, in fluid dynamics a Dirichlet boundary might indicate an incoming flow source.

***Neumann* boundary condition.** *Neumann* boundaries specify a constant gradient (derivative) across a boundary. For example, a constant derivative of 0 across a boundary is provided by mirroring the data inside the array to the boundary values. **Figure** 7.4 illustrates the effect of the Neumann boundary. In comparison with **Figure** 7.3 artefacts are much less pronounced. For image processing, Neumann boundaries are useful if speed is required and low-gain errors are acceptable. For fluid dynamics, Neumann boundaries can model reflective or absorptive surfaces (depending on the derivative constant).

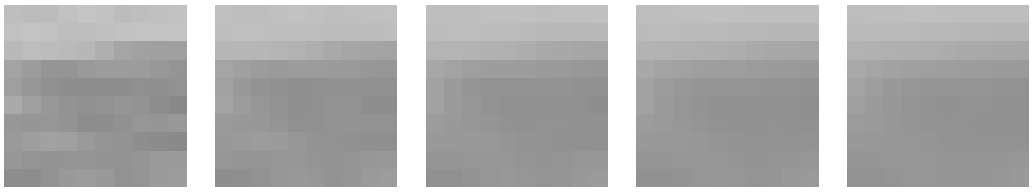
The *parameterised specific form* of the boundary syntax is parameterised by a parameter grid (which does not have any exterior regions) allowing an arbitrary term on the right-hand side involving a grid, where a general indexing operator (!!!) is allowed (this is the only place where (!!!) can be used; it is a function belonging only to the boundary syntax). Thus, a Neumann boundary can be defined in the following way, where this example defines a Neumann boundary for the lower edge of the X dimension:

$$
[\textbf{boundary} \mid X :* Y : (-1,j)\ g \to g\ !!!\ (0,j) \mid] \quad \text{-- } i.e.\ \frac{\partial f}{\partial x, y}(0,j) = 0.
$$



(a) i = 0 (b) i = 1 (c) i = 2 (d) i = 3 (e) i = 4

Figure 7.3. Bottom-right 10x10 sample for five iterations of the two-dimensional Gaussian on the *Lab* image, with Dirichlet boundaries ($f(n,m) = 0$, where (n,m) is a two-dimensional index in the image exterior). Artefacts caused by the default edge values can be clearly seen.



(a) i = 0 (b) i = 1 (c) i = 2 (d) i = 3 (e) i = 4

Figure 7.4. Bottom-right 10x10 sample for five iterations of the *gauss2D* function for the *CArray* comonad on the *Lab* image, with Neumann boundaries $\frac{df}{dx}(n, m) = 0$. Artefacts are significantly diminished compared with **Figure** 7.3.

Periodic boundary condition. Periodic boundary conditions map out-of-bounds indices to the opposite side of an array providing a *wrapping* behaviour. Periodic boundaries are useful for modelling connected surfaces of three-dimensional objects, *e.g.*, toroids.

A periodic boundary can be defined in the following way, where this example shows a periodic boundary in the X dimension, wrapping in both directions (where *extent* returns the bounds of a grid in a particular dimension, in this case X):

$$[\mathbf{boundary} \mid X :* Y : (-1, j)\ g \to g\ !!!\ ((\mathit{extent}\ X\ g) - 1, j) \\ (+1, j)\ g \to g\ !!!\ (0, j)]$$

Other boundary conditions. The boundary conditions described above are common, although there are other possible boundary conditions not mentioned here. A program may combine multiple boundary conditions, with different sides of the array having different conditions by using the various types of region descriptor discussed here.

Example 7.1.1. The following example combines some of the examples from above in a single Ypnos program computing the *difference of Gaussians* operation on a grid:

$$[\mathbf{dimensions} \mid X, Y \mid]$$

$$\mathit{gauss2D} = [\mathbf{stencil} \mid X :* Y : \mid \_\ \ t\ \ \_ \mid \\ \mid l\ \ @c\ r \mid \\ \mid \_\ \ b\ \ \_ \mid \Rightarrow (t + l + r + b + 2 * c)\ /\ 6.0 \mid]$$

$$\mathit{gaussDiff2D} = [\mathbf{stencil} \mid g \Rightarrow \quad x \leftarrow \mathit{gauss2D}\ g \\ y \leftarrow \mathit{gauss2D}\ x \\ (\mathsf{current}\ x) - (\mathsf{current}\ y) \mid]$$

$$\mathit{dirichlet1} = [\mathbf{boundary} \mid X :* Y : \mathbf{from}\ (-1, -1)\ \mathbf{to}\ (+1, +1) \to 0.0 \mid]$$

$$g = \mathsf{listGrid}\ (\mathsf{Dim}\ X :* \mathsf{Dim}\ Y)\ (0, 0)\ (w, h)\ \mathit{imgData}\ \mathit{dirichlet1}$$

$$g' = \mathsf{apply}\ g\ \mathit{gaussDiff2D}$$

Here *imgData*, w, h are free variables defined elsewhere, providing an image as a list of double-precision floating-point values, and the horizontal and vertical size of the image respectively.

7.1.5. Safety invariants

The coeffect information from grid patterns, describing the relative data access pattern of a stencil, is used to guarantee a *safety* invariant of stencil computations: that a stencil computation never causes an *out-of-bounds error*, or equivalently, every grid has sufficient exterior elements defined such that relative indexing is always defined.

This invariant is not only useful for verification, but also implies an optimisation: *at runtime, no bounds checking on array indexing is required.* The safety invariant and optimisation are guaranteed by encoding coeffects in Haskell's type system, described in Section 7.2 (p. 153).

7.1.6. Reductions

A common operation on containers is to perform a *reduction* of a container to a single value, such as computing the mean or maximum value. Ypnos provides two reduction primitives, reduceSimple and reduce. where reduceSimple takes a container of element type τ and an associative binary operator $\tau \to \tau \to \tau$.

Some reductions generate values of type different to the element type of the container being reduced. A Reducer structure packs together a number of functions for parallel reduction. The mkReducer constructor builds a Reducer, taking four parameters:

- A function reducing an element and partially-reduced value to a partially-reduced value: $(a \to b \to b)$
- A function combining two partially-reduced values: $(b \to b \to b)$
- An initial partial result: b
- A final conversion function from a partial-result to the final value: $(b \to c)$.

Figure 7.5(b) shows the types of these operations.

7.1.7. Iterative stencil application

A common program structure applies a stencil function repeatedly until a *convergence condition* is reached. This idiom is captured by the iterate function, which takes a stencil function, a *Reducer* of boolean result for the convergence condition, and a parameter grid. **Figure** 7.5(c) shows the type of iterate. The parameter grid and return grid must have the same element type for iterative stencil function application, as the result of one application is passed to the next.

In a functional setting, these operations are expensive since they involve repeated allocations of new arrays. An optimised implementation of iterate can be provided by using a *double buffering technique* where two mutable arrays are allocated, and an iteration reads from one and writes to the other. The roles of the arrays are then swapped for the next iteration (illustrated in **Figure** 6(a)). By the comonadic structure of stencil computations, writes never overlap/interfere

$$\mathsf{apply} :: (\mathsf{Container}\ c, \mathit{Inv}\ c\ a, \mathit{Inv}\ c\ b, \mathit{InvF}\ c\ a\ b) \Rightarrow (c\ a \to b) \to c\ a \to c\ b$$
$$\mathsf{current} :: (\mathsf{Container}\ c, \mathit{Inv}\ c\ a) \Rightarrow c\ a \to a$$

(a) Core (relative) comonadic operations

$$\mathsf{reduceSimple} :: (\mathsf{Container}\ c, \mathit{Inv}\ c\ a) \Rightarrow (a \to a \to a) \to c\ a$$
$$\mathsf{mkReducer} :: \forall\ b\ .\ (a \to b \to b) \to (b \to b \to b) \to b \to (b \to c) \to \mathit{Reducer}\ a\ c$$
$$\mathsf{reduce} :: (\mathsf{Container}\ c, \mathit{Inv}\ c\ a) \Rightarrow \mathit{Reducer}\ a\ b \to c\ a\ b \to b$$
$$\mathsf{iterate} :: (\mathsf{Container}\ c, \mathit{Inv}\ c\ a, \mathit{InvF}\ c\ a\ a) \Rightarrow (c\ a \to a) \to \mathit{Reducer}\ a\ \mathit{Bool} \to c\ a \to c\ a$$

(b) Reductions, and reduction-related, operations

$$\mathsf{iterateT} :: (\mathsf{Container}\ (\mathsf{Grid}\ d\ b), \mathit{Inv}\ (\mathsf{Grid}\ d\ b)\ a, \mathit{InvF}\ (\mathsf{Grid}\ d\ b)\ a\ a)$$
$$\Rightarrow \mathsf{Grid}\ (\mathsf{Dim}\ T :* d)\ b\ a \to a) \to \mathit{Reducer}\ a\ \mathit{Bool} \to \mathsf{Grid}\ d\ b\ a \to \mathsf{Grid}\ d\ b\ a$$
$$\mathsf{apply\_} :: (\mathit{Inv}\ (\mathsf{Grid}\ d\ b)\ x, \mathit{Inv}\ (\mathsf{Grid}\ d\ b)\ y, \mathit{Dimension}\ d) \Rightarrow$$
$$\Rightarrow (\mathsf{Grid}\ d\ b\ x \to y) \to \mathsf{Grid}\ d\ b\ x \to \mathsf{Grid}\ d\ (\mathit{Nil}, \mathit{Static})\ y$$

(c) Grid-specific operations

Figure 7.5. Types of Ypnos core operations

since a stencil function computes the value of a single element at a time and comonadic extensions applies the stencil function to every context exactly once.

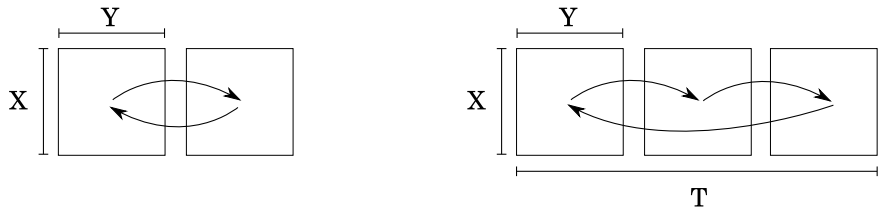
The iterateT operation for grids extends the (internally) destructive behaviour of iterate to computations involving further past versions of a grid, without the programmer having to alias grids between calls to apply. The iterateT primitive embeds a grid into a reserved temporal dimension T over which grid patterns are reused to define the exact number of previous grid versions required in a computation. Given this information, iterateT keeps in memory enough copies of the grid to satisfy this *historic* grid pattern. When previous grid allocations can no longer be accessed by the stencil function, iterateT safely reuses these allocations cyclically. For example, the following temporal stencil matches the two preceding iterations[1]:

$$T :|\ g''\ \ g'\ \ @\_\ |$$

Subsequently, iterateT allocates three mutable grids which are cyclically written to in-place (illustrated in **Figure** 6(b)). The static guarantees of grid patterns obviates the need for an alias analysis on iterateT (or system such as *linear types* [**Wad90b**]) to ensure that intermediate grids are not aliased and then used later after being destructively updated or deallocated; grid patterns provide decidable, compile-time information of the exact number of required grid allocations.

Neither iterate nor iterateT extend the expressive power of Ypnos, they are merely optimised forms of operations which could be derived using apply and reductions. The use of either operation communicates to the compiler that the programmer wishes to use the optimisation of mutable state.

[1]The cursor matches the current iteration's grid and must be a wildcard pattern, as iterateT marks the current grid version as undefined.



(a) iterate (b) iterateT [**stencil**| T : | g'' g' @_ | . . . |]

Figure 7.6. Allocation reuse patterns for iterate (double buffering) and iterateT with a temporal stencil two iterations into the past (triple buffering).

7.2. Implementation and mathematical structure

This section briefly details the mathematical structures behind Ypnos and its implementation in Haskell. The implementation and the type-level encodings of data access patterns and safety predicates is described in more detail in [**OM11**]. Type-level integers and lists are used in the implementation, which are encoded using GADTs in Haskell (described in Appendix E.1.1 (p. 221) and E.1.2 (p. 221)), with inductive definitions, *e.g.*, $S\ Z :: Nat\ (S\ Z)$ gives a type-level representation of the natural number 1 as a singleton type. For brevity, standard notation for lists and integers (*e.g.* [1,2,3]) will be used here, instead of inductive GADT-based encodings.

7.2.1. Grids and stencil application

The Grid data type in Ypnos comprises an unboxed array *UArray*, as well as additional structure describing its dimensionality, size, and boundaries. The Grid type constructor has three type parameters: Grid $d\ b\ a$, where d is a dimensionality type, b gives a type-level description of the grid's boundaries, and a is the element type of the grid. The data type is defined:

```
data Grid d b a where
  Grid :: (UArray (Index d) a)      -- Array of unboxed values
       → Dimensionality d           -- Dimensionality term
       → Index d                    -- Cursor ("current index")
       → (Index d, Index d)         -- Lower and upper bounds of extent
       → Boundary ixs d a           -- Boundary information
       → Grid d ixs a
```

The unboxed array (*UArray*) contains the values for both the interior and exterior elements of the array. *Index* is a type-level function that maps dimensionality to an *Int*-tuple type. The *Boundary* structure (described in the next section) encodes the boundary behaviour of the grid and is generated from a **boundary** macro. The type parameter *ixs*, which parameterises the resulting Grid type, encodes a type-level list of the boundary information.

Rather than using the Grid data constructor directly, the following constructor function can be used to construct a grid from a list of index-values pairs:

```
grid :: (IArray UArray a, Dimension d) ⇒
        Dimensionality d        -- Dimensionality term
        → Index d               -- Upper-bound index
        → Index d               -- Lower-bound index
        → [(Index d, a)]        -- Data (index-value pairs)
        → Boundary ixs d a      -- Boundary
        → Grid d ixs a
```

The listGrid constructor used in the previous section has the same type, modulo the data, which has type $[a]$ rather than a list of index-value pairs for grid.

Stencil application. Grid patterns are desugared via the **stencil** macro into relative indexing operations on a grid, with type approximately of the form:

$$index :: Safe\ i\ b \Rightarrow i \rightarrow \mathsf{Grid}\ d\ b\ a \rightarrow a$$

where *index* takes a relative index of type i and a grid of element type a, returning a single value of type a. The $Safe\ i\ b$ constraint enforces safety by requiring that there are sufficient boundary regions described by b such that a relative index i accessed from anywhere within the grid's extent has a defined value.

Stencils can be applied to grids using the apply operation, of type:

$$\mathsf{apply} :: (\mathsf{Grid}\ d\ b\ a \rightarrow a) \rightarrow \mathsf{Grid}\ d\ b\ a \rightarrow \mathsf{Grid}\ d\ b\ a$$

This type resembles that of *extension* for a comonad (see below), but restricts stencils to those that preserve the element type of the array. This restriction is required since the internal array of a grid contains both interior and exterior elements (in the boundary), of type a. A stencil operation is applied only over the interior elements by apply, where the exterior elements provide the boundary values for the operation. Thus, the stencil must preserve the element type of the grid ($\mathsf{Grid}\ d\ b\ a \rightarrow a$) to maintain type safety, otherwise the resulting grid would have exterior and interior elements of different types within the same array.

An additional *extension*-like operation is provided which computes a grid with a different element type than its parameter grid by discarding the boundary for the grid:

$$\mathsf{apply\_} :: (IArray\ UArray\ y) \Rightarrow (\mathsf{Grid}\ d\ b\ x \rightarrow y) \rightarrow \mathsf{Grid}\ d\ b\ x \rightarrow \mathsf{Grid}\ d\ [\,]\ y$$

The return grid has an empty list for its boundary-information annotation.

7.2.2. Boundaries

Ypnos boundary definitions are desugared by the **boundary** macro into a data structure *Boundary* which encapsulates functions for computing a grid's boundary. The *Boundary* type constructor is parameterised by various type-level information, including a type-level encoding of the descriptors for the boundary regions defined. This type-level boundary information is propagated to a grid's type (the b type parameter in the Grid type) when a grid is constructed.

The *Safe* constraint in the type of relative indexing operations uses this boundary information to ensure a grid has adequate boundaries for the relative index to be safe when accessed from anywhere within the grid's interior.

A grid whose boundary is defined without any parameterised regions has its exterior elements computed just once, as its values do not depend on a grid's inner values. This is called a *static boundary*. A grid whose boundary is defined with parameterised regions must have its exterior elements recomputed after application of a stencil to a grid; this is a *dynamic boundary*.

The *Boundary* type has three type parameters: $Boundary\ ixs\ d\ a$ where *ixs* is a type-level list of the boundary region descriptors (each with a constructor describing whether the boundary is dynamic D or static S), d is the dimensionality of the boundary definition, and a is the element type.

Rather than describing the full desugaring here, a few example boundary types are given to illustrate the type-level information parameterising *Boundary* (a thorough account of the desugaring and type-level computations is described in [**OM11**]).

$$\left[\!\!\left[\begin{array}{ll} X\colon & -1 \to 0.0 \\ & +1 \to 0.0 \end{array}\right]\!\!\right] :: Boundary\ [S\ (-1), S\ (+1)]\ (\mathsf{Dim}\ X)\ Double$$

$$\begin{aligned} &[\![X :\!*\ Y : \mathbf{from}\ (-1,-1)\ \mathbf{to}\ (+1,+1) \to 0.0]\!] \\ &\qquad :: Boundary\ [S\ (-1,-1), S\ (-1,*), S\ (-1,+1), S\ (*,-1), S\ (*,+1), \\ &\qquad\qquad S\ (+1,-1), S\ (+1,*), S\ (+1,+1)]\ (\mathsf{Dim}\ X :\!*\ \mathsf{Dim}\ Y)\ Double \end{aligned}$$

$$\begin{aligned} &\left[\!\!\left[\begin{array}{lll} X :\!*\ Y\colon & (-1,j)\ g \to g\ !!!\ (i,1) & \text{-- wrap bottom to top} \\ & (+1,j)\ g \to g\ !!!\ (i,-1) & \text{-- wrap top to bottom} \end{array}\right]\!\!\right] \\ &\qquad :: Boundary\ [D\ (-1,-1), D\ (-1,*), D\ (-1,+1)]\ (\mathsf{Dim}\ X :\!*\ \mathsf{Dim}\ Y)\ a \end{aligned}$$

where $*$ represents any index value. Note, these types are never constructed by an end-user.

7.2.3. (Relative) comonadic structure

Whilst the focus is on grids here, Ypnos is parameterised by any relative comonad (using the extended encoding of Section 5.1.3, p. 97) which is abstracted by the Container class:

$$\begin{aligned} &\mathbf{type}\ Inv\ c\ a = RObjs\ c\ a \\ &\mathbf{type}\ InvF\ c\ a\ b = RMorphs\ c\ a\ b \\ &\mathbf{class}\ RComonad\ c \Rightarrow \mathsf{Container}\ c\ \mathbf{where} \\ &\quad \mathsf{current} :: (Inv\ c\ a) \Rightarrow c\ a \to a \\ &\quad \mathsf{current} = counit \\ &\quad \mathsf{apply} :: (Inv\ c\ a, Inv\ c\ b, InvF\ c\ a\ b) \Rightarrow (c\ a \to b) \to c\ a \to c\ b \\ &\quad \mathsf{apply} = coextend \end{aligned}$$

The *Inv* and *InvF* types alias the previous constraint families of the relative comonad definition in Section 5.1.3 (p. 97). The operations of the relative comonad here are used to desugar the **stencil** notation analogously to the **codo**-notation (see Section 4.2, p. 83).

Grids. Grid is a relative comonad, with the following instance for *RComonad*:

```
instance (Dimension d) ⇒ RComonad (Grid d b) where
  type RObjs (Grid d b) x = IArray UArray x
  type RMorphs (Grid d b) x y = x∼y
  counit (Grid arr _ c _ _) = arr ! c
  coextend f (Grid arr d c (b1, b2) boundaries) =
    let es′ = map (λc′ → (c′, f (Grid arr d c′ (b1, b2) boundaries))) (range (b1, b2))
        arr′ = accum (curry snd) arr es″   -- preserve the boundaries
    in Grid arr′ d c (b1, b2) boundaries
```

This is similar to the comonad of pointed arrays using *Array* in Example 3.1.10 (p. 50) and the relative comonad of pointed unboxed arrays using *UArray* in Example 5.1.4 (p. 97). A key difference here though is that *coextend* applies the stencil function f at every index in the *interior* range, specified by *b1* and *b2*, rather than at ever index in the array; the exterior elements are not transformed. The *accum* function replaces the interior elements of the array *arr* with the new elements (es') whilst preserving the exterior elements of *arr*.

Using the categorical programming analogy, this structure is analogous to a relative comonad restricted to the subcategory of Haskell types that are instance of the *IArray UArray* class (those types whose values may be elements of an unboxed array). Furthermore, the above definition restricts the morphisms of the relative coKleisli category to be endomorphisms (discussed in Section 5.1.2, p. 95), such that only stencils of type Grid d b $a \to a$ can be applied, and thus apply has the type described above.

7.2.4. Coeffects and relative indexed comonad structure

Formally, the Grid relative comonad in Ypnos is understood as an *indexed relative comonad*; for d-dimensional grids (Grid d b) : $\mathcal{P}(\mathbb{Z}^d) \to [IArray\ UArray, IArray\ UArray]$, *i.e.*, (Grid d b) is a family of endofunctors on the *IArray UArray* subcategory of **Hask**, indexed by sets of d-dimensional indices. These sets of indices provide the data access coeffects arising from grid patterns, with the (first-order[2].) coeffect algebra structure $(\mathcal{P}(\mathbb{Z}^d), \cup, \emptyset, \cup)$.

These coeffects are encoded using Haskell type class constraints. This can be illustrated by *index* :: *Safe i b* ⇒ $i \to$ Grid d b $a \to a$ (whose type is an approximation of the actually indexing operations, described below in Section 7.2.5). Consider a stencil f with type:

$$f :: (Safe\ i\ b, Safe\ j\ b, Safe\ k\ b) \Rightarrow \mathsf{Grid}\ d\ b\ x \to y$$

[2]for *let*-binding (binding statements in the **codo**/**stencil** notation) only, *i.e.* without the $\sqcap$ operation

The stencil function f accesses relative indices i, j, k which must be safe with the respect to the boundary information b of the parameter grid, *i.e.*, there are adequate boundaries such that indexing i, j, k is defined anywhere in the interior. These constraints are the stencil's latent coeffect, *i.e.*

$$f :: \mathsf{Grid}\ d\ b\ x \xrightarrow{\{i,j,k\}} y$$

These data access coeffects are encoded using Haskell constraints for two reasons 1) grid patterns are static, therefore the coeffect of the data access pattern is statically decidable 2) the coeffect algebra is defined over sets with conjunction for its operations which can be modelled easily by type class constraints as described in Section 6.5.2 (p. 138). The *Safe* predicate implements the invariant implied by the coeffect (defined more precisely below).

7.2.5. Indexing and safety

Grid patterns are desugared into a number of relative indexing operations which are internally implemented without bounds checking. The one- and two-dimensional relative indexing operations have type (where $IntT$ is the type constructor for type-level integers):

$$\begin{aligned} &\mathit{index1D} :: \mathit{Safe}\ (\mathit{IntT}\ n)\ b \Rightarrow \\ &\qquad \mathit{IntT}\ n \to \mathit{Int} \to \mathsf{Grid}\ (\mathsf{Dim}\ d)\ b\ a \to a \\ &\mathit{index2D} :: \mathit{Safe}\ (\mathit{IntT}\ n, \mathit{IntT}\ n')\ b \Rightarrow \\ &\qquad (\mathit{IntT}\ n, \mathit{IntT}\ n') \to (\mathit{Int}, \mathit{Int}) \to \mathsf{Grid}\ (\mathsf{Dim}\ d :* \mathsf{Dim}\ d')\ b\ a \to a \end{aligned}$$

Note that, in both cases, the first parameter is a relative index and the second parameter is an absolute index. The **stencil** macro generates at compile time both an inductively defined relative index, to produce the correct type constraints, and an *Int*-valued relative index which is used to perform the actual access. This second parameter is provided so that an inductive relative index does not have to be (expensively) converted into an *Int* representation for actual indexing at run-time.

The *Safe* constraint provides a predicate enforcing safety by requiring that a grid's boundary provides sufficient elements outside of a grid such that a relative index has a defined value if accessed from anywhere within the grid. *Safe* is defined recursively as follows (for one dimension):

$$\begin{aligned} &\textbf{class}\ \mathit{Safe}\ i\ b \\ &\textbf{instance}\ \mathit{Safe}\ 0\ b \\ &\textbf{instance}\ (\mathit{Safe}\ (i-1)\ b, i\ \in\ b) \Rightarrow \mathit{Safe}\ i\ b \end{aligned}$$

where $\in$ here is a type-level list membership operation (see definition in Appendix E.1.2, p. 221). Thus, a relative index i accessed at the edge of a grid's interior must have a boundary region defining the exterior value at this index. For two-dimensions, the *Safe* predicate is defined:

$$\begin{aligned} &\textbf{instance}\ \mathit{Safe}\ (0,0)\ b \\ &\textbf{instance}\ (\mathit{Safe}\ (i-1,j)\ b, \mathit{Safe}\ (i,j-1)\ b, (i,j)\ \in\ b) \Rightarrow \mathit{Safe}\ (i,j)\ b \end{aligned}$$

(which generalises in the obvious way to n-dimensions).

The *Safe* type constraints of the indexing operation thus encode the coeffect of the stencil and constrain the grid's boundary regions, enforcing the invariant that the grid must *at least* satisfy the relative indices of the grid pattern.

As an example, the Gaussian operator of the introductory example is desugared by the **stencil** macro into the code shown in **Figure** 7(a) with the type shown in **Figure** 7(b) where inductive definition of relative indices is shown at the term-level where, for example, the value $(Neg\ (S\ Z), Pos\ (S\ Z))$ has its type represented as: $(-1, +1)$. Thus, the collection of *Safe* constraints encodes the coeffect of *gauss2D*.

$$
\begin{array}{l}
gauss2D\ g = \\
\quad \textbf{let}\ w = index2D\ (Neg\ (S\ Z), Pos\ Z)\ (-1,0)\ g \\
\qquad\ e = index2D\ (Pos\ (S\ Z), Pos\ Z)\ (1,0)\ \ g \\
\qquad\ s = index2D\ (Pos\ Z, Pos\ (S\ Z))\ (0,1)\ \ g \\
\qquad\ n = index2D\ (Pos\ Z, Neg\ (S\ Z))\ (0,-1)\ g \\
\qquad\ c = index2D\ (Pos\ Z, Pos\ Z)\ \ \ (0,0)\ \ g \\
\quad \textbf{in}\ (n + w + e + s + 2 * c)\ /\ 6.0
\end{array}
$$

(a) Code

$$
\begin{array}{l}
:: (Safe\ (-1,0)\ b, \\
\quad Safe\ (+1,0)\ b, \\
\quad Safe\ (0,+1)\ b, \\
\quad Safe\ (0,-1)\ b, \\
\quad Safe\ (0,0)\ b, Num\ a) \\
\Rightarrow \mathsf{Grid}\ (\mathsf{Dim}\ X :* \mathsf{Dim}\ Y)\ b\ a \to a
\end{array}
$$

(b) Type, with coeffect encoded by *Safe* constraints

Figure 7.7. Desugared *gauss2D* example

7.3. Results & analysis

A brief quantitative comparison of a prototype implementation of Ypnos versus Haskell is provided here with two benchmark programs of the two-dimensional Laplace operator and a Laplacian of Gaussian operator, which has a larger access pattern.[3]

All experiments are performed on a grid of size 512×512. The execution time of the whole program is measured for one iteration of the stencil and for 101 iterations of the stencil. The mean time per iteration is computed by the difference of these two results divided by a 100. The mean of ten runs is taken with all results given to four significant figures.

Laplace operator. This benchmark uses the discrete Laplace stencil.

	Haskell	Ypnos
1 iteration (s)	3.957	5.179
101 iterations (s)	6.297	7.640
Mean time per iteration (s)	0.0234	0.0246

Ratio of mean time per iteration for Ypnos over Haskell =

$$\frac{(7.640 - 5.179)/100}{(6.297 - 3.957)/100} = \frac{0.0246}{0.0234} \cong 1.051$$

i.e., Ypnos is approximately 5% slower per iteration than the Haskell implementation.

[3] All experiments were performed on an Intel Core 2 Duo 2Ghz, with 2GB DDR3 memory, under Mac OS X version 10.5.8 using GHC 7.0.1. All code was compiled using the `-O2` flag for the highest level of "non-dangerous" (i.e. no-worse performance) optimisations. The Ypnos library is imported into a program via an `#include` statement, allowing whole-program compilation for better optimisation.

Laplacian of Gaussian. The Laplacian of Gaussian operation combines Laplace and Gaussian convolutions. A 5×5 Laplacian of Gaussian convolution operator is used [**JKS95**]

	Haskell	Ypnos
1 iteration (s)	3.962	5.195
101 iterations (s)	7.521	8.662
Mean time per iteration (s)	0.0356	0.0347

$$\text{Ratio of mean time per iteration for Ypnos over Haskell} = \frac{0.0347}{0.0356} \cong 0.975$$

i.e. Ypnos is roughly 3% faster per iteration than the Haskell implementation.

Discussion. In terms of whole program execution time, Ypnos performance is slightly worse than Haskell, mostly due to the overhead of the general grid and boundary types and the cost of constructing boundary elements from boundary definitions. However, per iteration Ypnos performance is comparable to that of Haskell. In the Laplace example, performance is slightly worse, where the benefits of bounds-check elimination do not offset any overhead incurred by the Ypnos infrastructure. Given that safe indexing has an overhead of roughly $20-25\%$ over unsafe indexing, for the two-dimensional Laplace operator on a 512×512 grid, then the overhead of Ypnos in the Laplace experiment is roughly $25-30\%$ per iteration. For the Laplace of Gaussians example performance is however slightly better (roughly 3%). Given intensive data access the benefits of bounds-check elimination begin to overshadow any overheads from Ypnos. Given a larger program with more stencil computations we conjecture that performance of Ypnos could considerably surpass that of Haskell's.

Whilst Ypnos incurs some overhead it provides the additional benefits of static correctness guarantees, the comparable ease of writing stencil computations compared to Haskell or some other language, and the generality of extending easily to higher-dimensional problems.

7.4. Discussion: other containers

Whilst Ypnos is parameterised by an (indexed) relative comonad, as described in Section 7.2, Ypnos mainly supports the *grid* data structure with its grid patterns and boundary syntax. Grid patterns provided an abstraction over data access operations for a grid, simplifying programming. A coeffect analysis for data access was encoded and used to enforce safety. Ypnos can be generalised to other containers, with a coeffect analysis for data access by generalising the notion of *relative indexing* for a comonadic container. Relative indexing can be described in terms of *navigation operations* which define those contexts which are immediately "adjacent" to the cursor context.

For *zipper* data types, navigation operations are provided explicitly; *e.g.*, the *left*, *right*, *up* and *left*, *right* operations for the tree and list zippers respectively. Taken together, these operations define a *neighbourhood* of contexts adjacent to the current context. The navigation operations of a container-like data type are *complete* when they can be iterated to access *any* data value in a structure. For arrays, complete navigation operations access adjacent elements

to the current in orthogonal directions. For example, complete navigation operations for two-dimensional arrays with cursor (i, j) access values at $(i, j-1)$, $(i, j+1)$, $(i-1, j)$, and $(i+1, j)$, sometimes called the *north*, *south*, *west*, and *east* directions respectively.

For recursive data types, *pattern matching* provides an equivalent navigation principle, where unrolling a step of recursion is equivalent to iterating a navigation operation of a zipper.

Whilst arrays provide *rectilinear* structured grids, other applications use *triangular* structure grids, or *unstructured grids* comprising graphs of polygons, sometimes called *meshes* (see for example the dissertation of Smith [**Smi06**]). **Figure** 8(a) illustrates a Gaussian blur over a triangular-mesh comonad. Thus, triangular meshes has different navigation operations to a rectilinear mesh with only three directions, shown in **Figure** 8(b).

7.4.1. Neighbourhoods and coalgebras

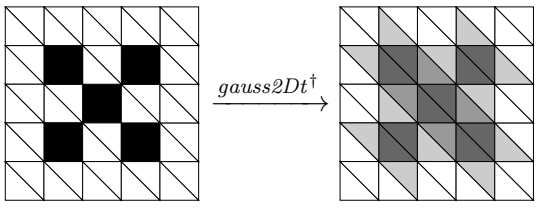
Navigation operations for a zipper such as *left*, *right*, or using pattern matching, or using indexing operations, all provide *deconstructors*, which can be described by the notion of a *coalgebra*.

Definition 7.4.1. A *coalgebra* (or F-coalgebra) for an endofunctor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ is a pair (U, c) of a *carrier object* $U \in \mathbb{C}_0$ and morphism $c : U \to \mathsf{F}U$ (the coalgebra operation).

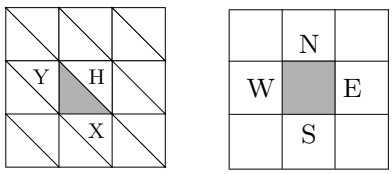
From the perspective of objects as types, a coalgebra $c : U \to \mathsf{F}\,U$ deconstructs some type U whose, structure is unknown, into components specified by the F endofunctor. A classic example is for the product functor $\mathsf{F}\,U = U \times A$ for some fixed A type of "output" values. A coalgebra $c : U \to U \times A$ takes U values to an observation of a value A and another U value, which might be thought of as observing an internal state ($\mathsf{observe} : U \to A$) composed in parallel with advancing an internal state ($\mathsf{next} : U \to U$).

The notion of a *final coalgebra* arises as the terminal object in a category of coalgebras, with morphisms defined by a morphism between carrier objects satisfying a coherence condition (see, *e.g.*, [**JR97**]). This aspect of coalgebras is not discussed here at length.

A final coalgebra gives a principle of definition by *coinduction*. For $\mathsf{F}\,X = A \times X$, the final coalgebra has as its carrier object the type $A^{\mathbb{N}}$ of streams on A with coalgebra morphism $\langle \mathsf{value}, \mathsf{next} \rangle s = (s0, \lambda n.\, s(n+1))$. For a coalgebra $(u, \langle \mathsf{observe}, \mathsf{next} \rangle)$, the coalgebra morphism to the final coalgebra is defined as $\lambda n.(\mathsf{observe} \circ \mathsf{next}^n) : U \to A^{\mathbb{N}}$.



(a) Example of a Gaussian blur on the triangular mesh comonad



(b) Relative indexing on rectilinear vs. triangular meshes

Figure 7.8. Illustration of a triangular mesh comonad

The navigating operations (deconstructors) of a non-empty list-zipper, of element type A, form a coalgebra for $\mathsf{F}U = A \times U \times U$:

$$\langle \mathsf{current}, \mathsf{left}, \mathsf{right} \rangle : \mathit{ListZipper}\, A \to A \times (\mathit{ListZipper}\, A) \times (\mathit{ListZipper}\, A)$$

where the first component of the product is the current element, and the second and third components are the list zippers to the left and right of the current, respectively.

Morphisms from a list-zipper coalgebra to its final coalgebra compute infinite binary trees, rather than lists, since there is a left and right path. List-zipper coalgebras can be given additional coherence conditions that regain the list-like behaviour:

$$(\mathsf{left} \circ \mathsf{right})\, x = x \qquad \textit{if } \mathsf{right}\, x \neq x$$
$$(\mathsf{right} \circ \mathsf{left})\, x = x \qquad \textit{if } \mathsf{left}\, x \neq x$$

Definition 7.4.2. For a functor F, with some notion of cursor such there is a *counit* $\varepsilon_A : \mathsf{F}A \to A$, and a set of n *complete navigation operations*, $\{f_i : \mathsf{F}A \to \mathsf{F}A \mid 1 \leqslant i \leqslant n\}$ a *neighbourhood coalgebra* is a coalgebra $(\mathsf{F}A, \mathsf{neighbourhood})$ where $\mathsf{N}\, A\, U = A \times \prod_{i=1}^{n} U$ is a bifunctor and $\mathsf{neighbourhood} = \langle \varepsilon, \langle f_1, \ldots, f_n \rangle \rangle : \mathsf{F}A \to \mathsf{N}\, A\, (\mathsf{F}A)$.

Thus a neighbourhood coalgebra returns a product of the current element with a product of the data structures focussed on its adjacent contexts.

Example 7.4.3 (Triangular mesh)**.** A neighbourhood coalgebra for triangular meshes has functor $\mathsf{N}AU = A \times U \times U \times U$ for current element and the mesh focussed on the three triangles adjacent to the cursor triangle.

Indexing schemes for polygon meshes have been proposed (*e.g.* the three component scheme of [**SZD+98**]), however such indexing schemes are relatively non-intuitive compared with standard, rectilinear mesh (*i.e.* array) indexing [**Smi06**]. The approach of defining a neighbourhood coalgebra for a data type provides indexing by repeated application of deconstructors.

The coeffect of data access can then be defined by associating a tag to each navigation operation and defining "relative indices" by lists of these tags. For example, the following defines two rules: [NAV] for navigating using the navigation operations of a neighbourhood coalgebra, constructing a relative index as a list of the navigation operations applied (denoted in the rule with a double question mark ??), and [VAL] for accessing the current value, which turns the relative index into a coeffect:

$$[\textsc{nav}]\frac{\Gamma \;??\; R \vdash e : U}{\Gamma \;??\; (n \,\mathsf{cons}\, R) \vdash (\pi_n \circ \mathsf{neighbourhood})\; e : U} \qquad [\textsc{val}]\frac{\Gamma \;??\; R \vdash e : U}{\Gamma \,?\, \{R\} \vdash (\pi_0 \circ \mathsf{neighbourhood})\; e : A}$$

Given a neighbourhood coalgebra N, a comonad can be automatically generated, which could be used to parameterise Ypnos. The following shows the general approach for constructing a comonad from a neighbourhood coalgebra in Haskell.

Special case. Consider a neighbourhood algebra of size 1, *i.e.*, it has the functor: $\mathsf{N}AU = A \times U$. Coalgebra morphisms for $\mathsf{N}A$, paired with a "seed", can be captured by the data type:

$$\textbf{data}\ \mathit{FCoAlg}\ a = \forall\ u\ .\ \mathit{FCoAlg}\ (u \to (a, u))\ u$$

This type describes all $(A \times -)$-coalgebra morphisms, where the carrier object (type) u is hidden from the outside. This data type is a comonad, with the following definition:

$$\begin{aligned}&\textbf{instance}\ \mathit{Comonad}\ \mathit{FCoAlg}\ \textbf{where}\\ &\quad \mathit{current}\ (\mathit{FCoAlg}\ f\ \mathit{seed}) = (\mathit{fst} \circ f)\ \mathit{seed}\\ &\quad \mathit{extend}\ k\ (\mathit{FCoAlg}\ f\ \mathit{seed}) = \mathit{FCoAlg}\ (\lambda \mathit{seed}' \to (k\ (\mathit{FCoAlg}\ f\ \mathit{seed}'), (\mathit{snd} \circ f)\ \mathit{seed}'))\ \mathit{seed}\end{aligned}$$

Thus, *current* applies the seed to the coalgebra morphism f, returning the "current" value (the *fst* component); *extend* lifts a morphism $k{:}\mathit{FCoAlg}\ a \to b$ to $\mathit{extend}\ k{:}\mathit{FCoAlg}\ a \to \mathit{FCoAlg}\ b$ by returning an $\mathit{FCoAlg}\ b$ value where at some carrier value seed', the new current value is calculate by applying k to the coalgebra with seed'. This comonad is equivalent to the composite comonad $\mathit{InContext}\ U \circ (- \times U)$ of *InContext* (with U contexts) and the product comonad, *i.e.*,

$$(\mathit{InContext}\ U \circ (- \times U))A = (U \Rightarrow (A \times U)) \times U$$

If a value of *FCoAlg* is the neighbourhood coalgebra streams $(A^{\mathbb{N}}, \langle \mathsf{current}, \mathsf{next} \rangle)$ (equivalent to the final coalgebra of $\mathsf{F}U = A \times U$), then the comonad is equivalent to the anti-causal streams comonad (suffix streams) of Section 3.2.4 (p. 56).

Generalising. This approach can be generalised to any neighbourhood coalgebra. In the following, the *flip* of a bifunctor N is written N°, where $\mathsf{N}^\circ\ U\ A = \mathsf{N}\ A\ U$.

Definition 7.4.4. Given a neighbourhood coalgebra on N, and a distributive law between $\mathsf{N}^\circ\ U$ and $\mathsf{InContext}\ U$ (costate), *i.e.*,

$$\begin{aligned}&\sigma_A : \mathsf{InContext}\ U\ (\mathsf{N}^\circ\ U\ A) \to \mathsf{N}^\circ\ U\ (\mathsf{InContext}\ U\ A)\\ \equiv\ &\sigma_A : \mathsf{InContext}\ U\ (\mathsf{N}\ A\ U) \to \mathsf{N}(\mathsf{InContext}\ U\ A)\ U\end{aligned}$$

then the *neighbourhood comonad* is the composite comonad $\mathsf{D} = (\mathsf{InContext}\ U) \circ (\mathsf{N}^\circ\ U)$, *i.e.*, $\mathsf{D}A = \mathsf{InContext}\ U\ (\mathsf{N}\ A\ U) = (U \Rightarrow (\mathsf{N}\ A\ U)) \times U$.

This construction defines a comonad since a neighbourhood coalgebra's bifunctor is of the form $\mathsf{N}AU = A \times \prod_{i=1}^{n} U$, *i.e.*, it is a product, and thus, $\mathsf{N}^\circ U$ has a product comonad structure. Since the product comonad distributes over the *in context* comonad, $(\mathsf{InContext}\ U) \circ (\mathsf{N}^\circ\ U)$ has the canonical composite comonad structure. The following defines the neighbourhood comonad construction in Haskell.

Example 7.4.5. Since $\mathsf{N}\ A\ U = A \times \prod_{i=1}^{n} U$, then a neighbourhood coalgebra is abstracted by the *Neighb* data type:

$$\textbf{data}\ \mathit{Neighb}\ m\ u\ a = \mathit{Neighb}\ a\ (m\ u)$$

where m abstracts the $\prod_{i=1}^{n} u$ component of N. To avoid cluttering the code with *flip* constructors, this data type already has its arguments flipped (*cf.*, $\mathsf{N}\,A\,U$). Thus, the bifunctor $\mathsf{N}\,U\,A = A \times U$ of stream coalgebras can be written **type** *Stream* $a = \forall\ u\ .\ \mathit{Neighb}\ \mathit{Id}\ u\ a$.

As discussed above, *Neighb* is simply an instance of the product comonad:

```
instance Comonad (Neighb f u) where
  current (Neighb x _) = x
  extend k (Neighb x ctx) = Neighb (k (Neighb x ctx)) ctx
```

Neighbourhood coalgebras are then captured by this data type (where : . composes data types):

```
data NCoAlg m a = ∀ u . NCoAlg (((CoState u) : . (Neighb m u)) a)
```

(recall, **data** *CoState* $u\ a$ = *CoState* $(u \to a)\ u$). *NCoAlg* is a comonad by the *in context* (costate) comonad (Section 3.2.6, p. 60) and comonad composition (Example E.2.7, p. 224):

```
instance Comonad (NCoAlg m) where
  extract (NCoAlg x) = extract x
  extend k (NCoAlg x) = NCoAlg (extend (k ∘ NCoAlg) x)
```

which unwraps the *NCoAlg* constructor and uses the underlying comonad instance for the composition of (*CoState* u) and (*Neighb* $m\ u$) provided by the distributive law:

```
instance ComonadDist (CoState u) (Neighb m u) where
  cdist (CoState f u) = let (Neighb _ ctx) = f u
                        in Neighb (CoState (λu' → let (Neighb x _) = f u' in x) u) ctx
```

Thus, *NCoAlg* values are neighbourhood coalgebras which deconstruct/observe a data structure (and may have some accompanying coherence conditions). If these are *final coalgebras* then the neighbourhood comonad corresponds to the *cofree comonad* (see Section 3.2.8, p. 61) where $\mathsf{D}A = \nu U.A \times \mathsf{N}\,A\,U$ where the fixed-point computes the usual coinductive definition of a final coalgebra. This approach therefore provides a way to generate comonads from the navigation operations of some data type (be it a zipper, or otherwise), defined as a neighbourhood algebra. This can be coupled with a coeffect system for data access, as described above, for analysing the navigation operations. Further work is to explore this approach further and integrate it into the Ypnos implementation.

7.5. Conclusion

This chapter applied the work of the previous chapters, defining a language for programming with containers, primarily *grids* (essentially arrays). The language was based on the **codo**-notation with the addition of the *grid pattern* syntax for abstracting over the navigation operations of an array and the *boundary* syntax for simplifying the definition of boundary values for a grid.

Related work. There are various other approaches to stencil programming in Haskell, for arrays, that are related to Ypnos, including PASTHA [**Les10**] and Repa [**KCL+10, LK11**]. Repa provides a library of higher-order *shape polymorphic* array operations for programming parallel regular, multi-dimensional arrays in Haskell [**KCL+10**]. Type-level representations of shape (dimensionality) are used which are similar to the dimensionality types of Ypnos. Stencils functions can be defined in Repa with data types abstracting stencils of a particular size with a function from indices within the stencil's range to values. The current implementation allows a constant or wrapped boundary to be specified which permits a bounds-check free implementation of a stencil application function. Ypnos differs to Repa in its type-level encoding of coeffects describing the data access pattern of stencil, which provides a more flexible language than Repa.

PASTHA is a similar library for parallelising stencil computations in Haskell [**Les10**]. Stencils in PASTHA are described via a data structure and are restricted to relative indexing in two spatial dimensions but can also index the elements of previous iterations. In comparison, Ypnos provides a more general approach with safety and optimisation guarantees following from its coeffect system.

There are many other languages designed for fast and parallel array programming such as Chapel [**BRP07**], Titanium [**YHG+07**], and Single Assignment C (SAC) [**Sch03**]. Ypnos differs from these in that it is sufficiently restricted to a problem domain, and sufficiently expressive within that problem domain, that all information required for guaranteeing safety, optimisation, and parallelisation is decidable and available statically without the need for complex compiler analysis and transformation.

7.5.1. Further work

Further optimisations. There has been much work on the optimisation of stencil computations particularly in the context of optimising cache performance (see for example [**KDW+06**]) and in the context of parallel implementations [**DMV+08, KBB+07**]. Ypnos does not currently use any such optimisation techniques but could be improved by the use of more sophisticated implementations of the stencil application functions. Various optimisation techniques such a loop tiling, skewing, or specialised data layouts, could be tuned from the symbolic information about access patterns statically provided by grid patterns.

Generalising grid patterns. Further work is to generalise the grid pattern syntax of Ypnos to other data structures. Section 7.4 defined containers in terms of navigations forming a *neighbourhood coalgebra*, describing the contexts immediately adjacent to the current context. This could be used to provide a data-type generic pattern syntax.

Chapter 8

Conclusions and further work

8.1. Summary

The broad aim of this dissertation has been to develop new programming language techniques to improve *the four Rs* of effective languages: *reading* (understandability), *writing* (productivity), *reasoning*, and *running* (performance), introduced in Chapter 1. Given the difficulty of maximising all of these attributes in a general setting, this dissertation focussed on *contextual* computations.

A typed and category theoretic approach was followed in both programming and semantics. Types were used to describe a computation's structure and to give an approximate specification of *what* is computed (in the case of standard type theory) and *how* it is computed (using *indexing*, in Chapter 5-7). From a typed perspective on programs, category theory was used to structure programs and to systematically define language semantics and their axiomatisation.

The contributions of this dissertation are summarised here in two ways: in terms of the mathematically structures used in programming and semantics in Section 8.1.1, and from the perspective of the improving the *four Rs* for contextual computations in Section 8.1.2

8.1.1. Categorical programming and semantics of contextual computations

The abstractions of category theory expose common structure in mathematics. The philosophy of seeking abstractions in category theory is paralleled in computer science and programming, where abstractions are sought to manage complexity and reduce work. Thus category theory provides a useful formalism for abstracting both programs and semantics.

Categorical semantics models a language using category theory. Chapter 2 presented a version of Lambek and Scott's CCC model for the simply-typed λ-calculus [**Lam80, LS88**] where type derivations are mapped to morphisms of a category, where $[\![v_1 : \tau_1, \ldots v_n : \tau_n \vdash e : \tau]\!] : (\tau_1 \times \ldots \times \tau_n) \to \tau \in \mathbb{C}_1$. Chapter 2 distinguished between the category theory structures required to give the semantics of simply-typed λ-calculus terms, and the additional structure/axioms which provide $\beta\eta$-equality. This approach was followed in Chapters 3 and 6 to give the semantics of *contextual* languages.

Categorical programming applies category theory to structure, abstract, organise, and reason about programs. This approach was discussed in Section 2.4, and used in Chapters 3, 4, and 7 for programming contextual computations. Chapters 4 and 7 developed contextual languages embedded into Haskell, whose desugaring is akin to the categorical semantics of Chapter 3.

Comonads. Contextual computations were defined by their encoding as morphisms/functions of the form $\mathsf{D}A \to B$ where a functor/data type D provides a representation of the *context*. Comonads provided the basis for a categorical semantics to a *contextual* λ-calculus in Chapter 3

(Section 3.3), following the approach described in Chapter 2, where terms are mapped to morphisms of a coKleisli category: $[\![\Gamma \vdash e : \tau]\!] : \mathsf{D}(\tau_1 \times \ldots \times \tau_n) \to \tau \in \mathbb{C}_{\mathsf{D}1}$. The semantics requires additional lax and colax monoidal structures for D, as shown by Uustalu and Vene [**UV08**]. The semantics in Chapter 3 is akin to Uustalu and Vene's semantics, but provides a generalisation and makes precise the necessary conditions for $\beta\eta$-equality (Section 3.3.2). Thus, Chapter 3 provided a scheme for understanding the syntactic properties of a contextual language based on the underlying mathematical structures.

Section 3.4 showed the limitations of the comonadic approach, notably *shape preservation* (Lemma 3.4.2, p. 71). Shape preservations implies that context encoded by a comonadic value $\mathsf{D}X$ is *uniform* throughout a computation. This restricts the range of computations captured by comonads.

Chapters 3, 4, and 5 defined comonads for various data types in Haskell, including: lists, streams, trees, arrays, graphs, and meshes. Data types with *cursors* (marking the current context) were shown in two styles: using *zippers* (*à la* Huet [**Hue97**]) and pointers (as in the *in context* comonad, $\mathsf{D}A = (C \Rightarrow A) \times C$). Stream, list, and tree comonads appear in the literature (*e.g.* [**UV06, UV07**]), whereas arrays, meshes, and graphs are new in this work (Chapter 3). Some of the presentations are also new, for example, the *pointed tree comonad* does not appear in the literature.

Chapter 4 contributed the **codo**-notation to simplify programming with comonads, where a **codo**-block defines a coKleisli morphism as a program:

$$(\textbf{codo}\ p\ \Rightarrow\ \overline{p \leftarrow e};\ e) :: \mathit{Comonad}\ c \Rightarrow c\ t \to t'$$

Indexed comonads. Chapter 5 developed a number of generalisations to comonads, where the encoding of context is refined by the *contextual requirements* of a computation. For example, the data access pattern of a local operation $f : \mathsf{D}A \to B$ has a contextual requirement that data accesses must be defined at all contexts. Thus, an argument to f must satisfy these requirements. In the comonadic approach, these requirements are implicit, encoded within a local operation's term. In Chapter 5, requirements were made explicit by indexed families of functors with contextual computations as morphisms of the form $\mathsf{D}_R X \to Y$.

Section 5.4 developed the novel *indexed comonad* structure whose operations are indexed by contextual requirements describing the propagation of requirements through a computation. **Figure** 8.1 summarises the structure. Indexed comonads relax the shape preservation property of comonads, so that the shape of the encoded context may change throughout a computation. However, shape may not change arbitrarily, but is governed by the monoidal structure on the indices $(\mathbb{I}_0, \bullet, I)$, shown in the laws of the indexed comonad.

Chapter 6 describes the contextual requirements of a computation as *coeffects*, defining a class of static analyses for coeffects, called *coeffect systems*. This approach dualises the *effect systems* originally by Gifford and Lucassen [**GL86**], with judgments of the form $\Gamma\ ?\ R \vdash e : \tau$ for an expression e with contextual requirements R. The general coeffect system is parameterised by a *coeffect algebra* $(C, \bullet, I, \sqcup, \sqcap)$ which provides operations for combining coeffects for the

– $\mathsf{D} : \mathbb{I} \to [\mathbb{C}, \mathbb{C}]$ with monoidal cat. $(\mathbb{I}, \bullet, I)$
– $(\varepsilon_I)_A : \mathsf{D}_I A \to A$
– $(\delta_{R,S})_A : \mathsf{D}_{R \bullet S} A \to \mathsf{D}_R \mathsf{D}_S A$
– $(-)^\dagger : (\mathsf{D}_S A \to B) \to \mathsf{D}_{R \bullet S} A \to \mathsf{D}_R B$

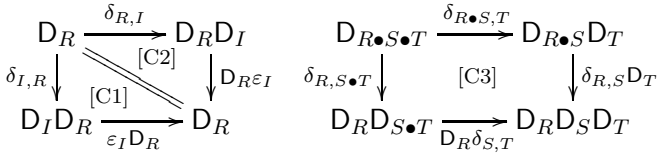


Figure 8.1. Summary of the indexed comonad structure

terms of the simply-typed λ-calculus. The indexed comonad structure of Chapter 5 was used in Chapters 6 to give a semantics to any contextual calculus with an associated coeffect system, with denotations:

$$[\![v_1 : \tau_1, \ldots, v_n : \tau_n \,?\, R \vdash e : \tau]\!] : \mathsf{D}_R([\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]) \to [\![\tau]\!]$$

Analogous monoidal structure for indexed comonads was developed in Section 6.4, where $\sqcup$ combines the coeffects for the colax operation $\sqcap$ for the lax operation.

An equational theory was developed in Section 6.3.1 (p. 121) and Section 6.4.3 (p. 130), where it was shown that syntactic axioms on a coeffect algebra correspond to semantic axioms on the model (see Remark 6.4.5, p. 131).

Chapter 7 used the coeffect approach in Haskell with the indexed *Grid* comonad (an array with interior and exterior elements), where the coeffect indices were encoded by type class constraints following the approach in Section 6.5.

8.1.2. Improving the *four Rs* for contextual computations

Chapter 1 broadly described actions on programs that an effective language should assist: *reading*, *writing*, *reasoning*, and *running*. These four tenets are non-orthogonal, for example a languages whose programs are easily reasoned about automatically may be amenable to automatic optimisations in a compiler. The aim of this dissertation has been to improve the *four Rs* of programming contextual computations. The following summarises the work from this perspective.

Reading and writing. Chapter 4 introduced the **codo**-notation which considerably simplified programming contextual computations with comonads, thus aiding reading and writing of contextual computations.

The Ypnos language introduced in Chapter 7 uses the **codo**-notation as well as the novel *grid patterns* syntax which simplifies writing stencil computations. The pictorial nature of the grid patterns makes it easier to understand a stencil's access pattern, avoiding a "soup" of indexing operations. Furthermore, grid patterns help the programmer to avoid programming errors through indexing mistakes, which are easy to introduce and hard to spot.

Reasoning. There has been an emphasis on equational theories throughout for the contextual languages (calculi) introduced. Chapter 2 discussed the basic equational theory of the simply-typed λ-calculus which was used as the basis for equational theories for the contextual λ-calculus

in Chapter 3 and for the coeffect calculus in Chapter 6. The additional axioms on comonads with a monoidal functor that are necessary to provide $\beta\eta$-equational theory were studied in depth in Section 3.3, and similarly for indexed comonads and indexed monoidal functors in Section 6.3.1 and Section 6.4.3.

The coeffect analysis described in Chapter 6 reconstructs the contextual requirements of a program which can be used to reason about its behaviour and correctness. This was was used in Chapter 7 for Ypnos, encoding coeffects as type class constraints, to enforce safe indexing operations. This provides the guarantee that *well typed programs can't go out of bounds*.

Running. Improved reasoning about programs also benefits their efficient execution. This was seen in Chapter 7, where the safety guarantees of Ypnos allowed the optimisation of bounds-check elimination on indexing operations, improving performance.

Section 4.2.1 (p. 85) showed how the laws of a comonad can be used to improve the asymptotic complexity of a program, where associativity of coKleisli extension: $(g \circ f^\dagger)^\dagger = g^\dagger \circ f^\dagger$ provides a rewrite rule from an $\in \mathcal{O}(n|g| + n^2|f|)$ program to an $\in \mathcal{O}(n|g| + n|f|)$ program.

Section 5.1 showed the *relative comonad* structure, the dual of *relative monads* by Altenkirch *et al.* [**ACU10, ACU11**], where a comonad is generalised from an endofunctor to a functor. This dissertation contributed an encoding of relative comonads in Haskell which, coupled with a *subcategory* interpretation of ad hoc polymorphism, allows comonad-like structures on data types whose operations are ad-hoc polymorphic. Those data types which have ad-hoc polymorphic operations tend to have efficient type-specific implementations. For example, the *Set* data type in Haskell is restricted in its polymorphism to types whose values are orderable, using an efficient balanced-binary tree implementation. Thus relative comonads facilitate efficient implementations of contextual computations. Chapter 7 used an *indexed relative comonad* with a coKleisli category of *endomorphisms* providing the *grid* comonad which provides an efficient solution to defining boundary values for a stencil computations.

Chapter 6 also showed how indexed comonads can be used to define *optimising semantics*. For example, the *indexed partiality comonad* (Example 5.4.2, p. 105) provides a semantics which automatically eliminates dead-code, where the coeffect algebra computes an approximate live-variable analysis. The *indexed stream comonad* (Example 6.4.8, p. 135) gave a semantics which calculates the minimum amount of elements to cache/buffer in a dataflow-style language.

8.2. Limitations and further work

Limitations and subsequent further work were considered in each chapter. The notable further work suggested is briefly summarised here followed by some more general limitations and suggestions of further work.

Summary of further work mentioned throughout. For the **codo**-notation in Chapter 4 rewrite rules in Haskell were described based on the comonad laws, which improve the asymptotic complexity of programs written using the notation (Section 4.2.1). Further work is to provide further rewrites for improving the complexity.

The general coeffect system of Chapter 6 approximates coeffects over the whole context of free variables. For example, the liveness analysis tracks whether any variable is live, rather than specific variables. Section 6.6.2 suggested a generalisation to *structural coeffects* where coeffects are tracked per-variable.

The Ypnos language is described generally for (relative) container comonads in Chapter 7. However, it primarily supports computations over arrays (called grids). Further work is to extend Ypnos to other containers, following from the approach of abstracting the *neighbourhood operation* of a container describing those contexts adjacent to the current, described in Section 7.4. Further work is to extend the grid pattern and boundary syntax to other containers, or to a data-type generic setting.

8.2.1. Recursion

This dissertation did not provide the necessary additional structure to give a semantics for recursion in contextual languages. A fixed-point can be added to the simply-typed λ calculus if there is a natural transformation $\mathsf{fix}_A : (A \Rightarrow A) \to A$ in the underlying category. The typing and semantics is then:

$$[\textsc{fix}]\frac{[\![\Gamma \vdash e : \tau \Rightarrow \tau]\!] = f : \Gamma \to (\tau \Rightarrow \tau)}{[\![\Gamma \vdash \mathsf{fix}\, e : \tau]\!] = \mathsf{fix}_\tau \circ f : \Gamma \to \tau}$$

For the contextual λ-calculus, the equivalent structure in a coKleisli category is therefore the natural transformation $\mathsf{fix}^{\mathsf{D}}_A : \mathsf{D}(\mathsf{D}A \Rightarrow A) \to A$, with the semantics:

$$[\textsc{fix}]\frac{[\![\Gamma \vdash e : \tau \Rightarrow \tau]\!] = f : \mathsf{D}\Gamma \to (\mathsf{D}\tau \Rightarrow \tau)}{[\![\Gamma \vdash \mathsf{fix}\, e : \tau]\!] = \mathsf{fix}^{\mathsf{D}}_\tau \circ^{\mathsf{D}} f : \mathsf{D}\Gamma \to \tau = \mathsf{D}\Gamma \xrightarrow{f^\dagger} \mathsf{D}(\mathsf{D}\tau \Rightarrow \tau) \xrightarrow{\mathsf{fix}^{\mathsf{D}}} \tau}$$

Uustalu and Vene semantics for Lucid used Haskell's own fixed-point semantics for recursion [**UV06**]. Similarly, for categorical programming with comonads, Section 4.5 (p. 88) embedded first-order Lucid into Haskell using **codo**-notation, reusing Haskell's fixed-point semantics to defined recursive coKleisli morphisms. Alternatively, a fixed-point combinator can be used which differs to the above fix, of type: $\mathsf{fix}' : (\mathsf{D}\tau \Rightarrow \tau) \to \mathsf{D}\tau$, with a default definition $\mathsf{fix}'\, f = f^\dagger(\mathsf{fix}'\, f)$.

An alternative useful fixed-point operator could be a *parameterized* fixed-point (see, *e.g.*, the work of Simpson [**SP00**]), on the coKleisli category, with:

$$[\textsc{pfix}]\frac{[\![\Gamma \vdash e : (\sigma \times \tau) \Rightarrow \tau]\!] = f : \mathsf{D}\Gamma \to (\mathsf{D}(\sigma \times \tau) \Rightarrow \tau)}{[\![\Gamma \vdash \mathsf{pfix}\, e : \sigma \Rightarrow \tau]\!] = \mathsf{pfix}_{\sigma,\tau} \circ f : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau)}$$

These additional operators were not considered here and their full exploration is further work.

Recursion and coeffects. The coeffect calculus of Chapter 6 also omitted recursion/fixed points. For effect systems, Wadler and Thiemann's include a rule for recursive binding [**WT03**]:

$$[\textsc{rec}]\frac{\Gamma, f : \sigma \to \mathsf{M}^F\tau, x : \sigma \vdash e : \mathsf{M}^F\tau}{\Gamma \vdash \mathbf{rec}\, f\, .\, \lambda x\, .\, e : \mathsf{M}^\perp(\sigma \to \mathsf{M}^F\tau)}$$

where the effect of the bound expression becomes the latent effect of the recursive function. Since effects are combined by a semilattice (which is idempotent) repeated effects are not captured.

For coeffects the situation is less straightforward. Consider the following (non-terminating) recursive program for dataflow $\mathsf{fix}\,(\lambda v.\mathsf{prev}\ v)$. The standard small-step semantics for fix is:

$$\mathsf{fix}\,(\lambda v.e) \rightsquigarrow \mathbf{let}\ v = \mathsf{fix}\,(\lambda v.e)\ \mathbf{in}\ e \tag{67}$$

then the example program computes $\lambda v.\ldots.\mathsf{prev}(\mathsf{prev}(\mathsf{prev}\ v)) = \lambda v.\mathsf{prev}^{\omega}\, v$. The body of the recursive function can be given the coeffect/typing:

$$\frac{\dfrac{\Gamma, v : \tau\,?\,0 \vdash v : \tau}{\Gamma, v : \tau\,?\,1 \vdash \mathsf{prev}\ v : \tau}\,[\textsc{prev}] \qquad \Gamma\,?\,F \vdash \mathsf{fix}\,(\lambda v.e) : \tau}{\Gamma\,?\,1 + F \vdash \mathbf{let}\ v = \mathsf{fix}\,(\lambda v.e)\ \mathbf{in}\ \mathsf{prev}\ v : \tau}\,[\textsc{let}]$$

where F is a placeholder for the coeffect of fix. By reduction (67), $F = 1 + F \Rightarrow F = \omega$.

For any coeffect algebra, this generalises to the equation $F = R \sqcup (R \bullet F)$ where R is the coeffect of the body of fix. A coeffect system with a fixed-pointed operation, or recursive binding, therefore requires an additional operation for the fixed-point of coeffects $\mathit{fix}\, R = \lim_{n\to\infty} \bigsqcup (\sum_n R)$ (derived from distributivity of $\bullet$ over $\sqcup$) and the rule:

$$\frac{\Gamma\,?\,S \vdash e : \tau \xrightarrow{R} \tau}{\Gamma\,?\,S \bullet (\mathit{fix}\, R) \vdash \mathsf{fix}\ e : \tau}\,[\textsc{fix}]$$

For some effect systems this will require limit elements, *e.g.* $\omega \in C$ for dataflow with additional rules for $+$ that $r + \omega = \omega$. For coeffect systems with idempotent $\sqcup$ and $\bullet$ operators the behaviour is simply that $\mathit{fix}\, F = F$. For example, liveness has: $\mathit{fix}\mathsf{F} = \mathsf{F} \sqcup (\mathsf{F} \bullet \mathsf{F}) \sqcup \ldots = \mathsf{F}$ and $\mathit{fix}\,\mathsf{T} = \mathsf{T} \sqcup (\mathsf{T} \bullet \mathsf{T}) \sqcup \ldots = \mathsf{T}$.

An extension to the general coeffect system, with a proper axiomatisation of the recursion and fixed-points, and extending of the contextual semantics is further work.

8.2.2. Mathematical structuring

In Chapter 5, the operations $\delta_{R,S} : \mathsf{D}_{(R \sqcup S)} \dot{\to} \mathsf{D}_R\mathsf{D}_S$ and $\varepsilon_{\perp} : \mathsf{D}_{\perp} \to 1_{\mathbb{C}}$ of an indexed comonad were derived in Section 5.4 from a *colax monoidal structure* between strict monoidal categories $(\mathbb{I}^{\mathsf{op}}, \sqcup, \perp)$ and $([\mathbb{C},\mathbb{C}], \circ, 1_{\mathbb{C}})$. Section 6.4 introduced additional operations akin to the lax and colax functor operations:

$$\mathsf{m}^{R,S}_{A,B} : \mathsf{D}_R A \times \mathsf{D}_S B \to \mathsf{D}_{(R \sqcap S)}(A \times B)$$
$$\mathsf{n}^{R,S}_{A,B} : \mathsf{D}_{(R \sqcup S)}(A \times B) \to \mathsf{D}_R A \times \mathsf{D}_S B$$

Further work is to abstract these operations, which at the moment have been added in an ad hoc manner, rather than systematically derived like the $\varepsilon_{\perp}$ and $\delta_{R,S}$ operations.

8.2.3. Modal logic

This dissertation focussed mainly on categorical models for languages, and did not consider the logically correspondences of these languages/categorical models.

Bierman and de Paiva developed a natural deduction system for intuitionistic S4 logic, modelling the necessity modality $\Box$ as a comonad, where the traditional axioms of S4 are modelled

by the counit and comultiplication of a comonad respectively [**BdP96**, **BdP00**]:

$$[T] \quad \Box A \to A$$
$$[4] \quad \Box A \to \Box\Box A$$

Bierman and de Paiva's model also includes a lax monoidal functor (with $\mathsf{m} : \Box A \times \Box B \to \Box(A \times B)$) such that hypothesis can be turned into implications with [**BdP96**], which follows the same use for the lax monoidal functor in the semantics of Chapter 3. Their work includes a corresponding term language (also similar to the work of Pfenning and Wong [**PW95**]) which resembles the contextual λ-calculus described in Chapter 3. Their approach separates introduction and elimination of $\Box$, which are not explicit in the λ-calculus approach.

Nanevski *et al.* develop a *contextual modal type theory* which specifically links modal logic with contextual notions in logic and computer science [**NPP08**]. This approach is used to structure *staged computations*, which was previously shown by Davies and Pfenning using a modal logic approach.

Further work is to consider the logic correspondences of the contextual calculi of the coeffect calculus and indexed comonads, as well as the relation between Kripke semantics of modalities and comonads.

Kripke semantics. *Kripke frames* provide a semantics for modal logic, comprising a pair $\langle W, R\rangle$ of a non-empty set of possible worlds W and an *accessibility relation* R between worlds [**Fit09**]. Properties of R correspond to axioms of a particular modal logic. For S4, these are:

$$\begin{array}{lll} [T] & \Box A \to A & \Leftrightarrow \forall w.(wRw) \\ [4] & \Box A \to \Box\Box A & \Leftrightarrow \forall w, u, v.(wRv \wedge vRu) \Rightarrow wRu \end{array}$$

corresponding to reflexivity and transitivity of the accessibility relation R.

As described in Chapter 3, the comultiplication operation of a comonad describes accessibility between contexts. For example, for the suffix list comonad, where position in the list are the context, only those contexts that are greater than the current are accessible. Thus, a Kripke frame $\langle \mathbb{N}, R\rangle$ is defined where R has the property: $\forall x \forall y.\ x \leqslant y \Rightarrow xRy$.

Via the Curry-Howard correspondence (proofs-as-terms, formulae-as-types) the necessity modality can be considered a data structure. Further work is to explore the correspondence between the a Kripke frame semantics for a modal logic and its comonad definition. This may provide a verification technique for comonad definitions, verifying comultiplication by checking transitivity and reflexivity of the accessibility relation, or to guide the process of defining a comonad.

Modal logic and distributed computation. Section 6.2.3 described a variant of the coeffect calculus for dynamically-bound resources which can be used in a distributed setting. There have been a number of languages in the literature which use modal types for distributed languages. For example, Murphy *et al.* described a λ-calculus (Lambda 5) for distributed computing based on an intuitionistic S5 modal logic where possible worlds are taken as nodes on a network and

therefore $\Box A$ is the type of a mobile computation that can be executed anywhere [**MVCHP04**]. Whilst comonads provide a model for the $\Box$-modality, Lambda 5 differs significantly to the contextual λ-calculus, where $\Box$ is a type constructor of the language (rather than implicit in the semantics) and can be introduced and eliminated by terms in the languages. Lambda 5 follows the approach of Simpson where possible words are explicit in judgments [**Sim94**]. The propagation of possible worlds however does not correspond to a coeffect system. Lambda 5 provides the basis for the ML5 language [**MCH08**].

Further work is to extend the coeffect approach to distributed resources into a full language, and to assess the applicability of the coeffect calculus to existing work in this field.

8.2.4. Comonads for natural language semantics

In the tradition of Montague, the syntax and semantics of natural languages can be treated mathematically in the same way as artificial languages in logic and programming [**Mon70**]. The (typed) λ-calculus is a popular model for languages, both natural and artificial.

Following the Montague tradition, and the approach of categorical semantics, monads have had some use in natural language semantics for structuring non-determinism, focus, variable binding and *intensionality* [**Sha02**]. Intensional contexts in sentences, such as "the student knows that..." [**Sha02**] and contextual parameters such as the *hearer* of a sentence [**Bek09**] have been described using the reader monad ($\mathsf{M}A = X \Rightarrow A$) (which was shown equivalent to the product comonad in Section 3.5, p. 75) and the state monad.

As described in the introduction, the *intension* of a phrase refers to the entire meaning, or *sense* as Frege put it, in all contexts, whereas *extension* is a particular (context-independent) denotation of a phrase [**Car47**]. In the tradition of Carnap, Bresson, Montague, and others, intensional objects may be treated as functions mapping from states, or *possible worlds* to extensions (values), *i.e.*, $w \to a$ maps possible worlds or states, or some contextual information (contexts) w to extensional values of type a. This was related to comonadic values of type $\mathsf{D}A$ for the exponent comonad in this dissertation. Relatedly, Shan used the reader monad for capturing notions of intensionality in natural language.

More recent works have described intensional semantics where intensions are parameters to computations [**vFH08**], describing semantics with morphisms roughly of the form $(x \to a) \to b$. Thus, it seems that these semantics can be abstracted by the use of a comonad (the exponent comonad in this case) to provide *intensional objects* and the semantics of composition for intensionality in natural language.

Further work is to apply the comonadic semantics approaches discussed in this dissertation to natural semantics, in collaboration with computational linguistics.

Bibliography

[AAG05] M. Abbott, T. Altenkirch, and N. Ghani, *Containers: constructing strictly positive types*, Theoretical Computer Science **342** (2005), no. 1, 3–27.

[AAMG04] M. Abbott, T. Altenkirch, C. McBride, and N. Ghani, *∂ for data: differentiating data structures*, Fundamenta Informaticae **65** (2004), no. 1-2, 1–28.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, et al., *The Landscape of Parallel Computing Research: A View from Berkeley*, Tech. Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ABD+08] Krste Asanovic, Ras Bodik, Demmel, et al., *The Parallel Computing Laboratory at U.C. Berkeley: A research agenda based on the Berkeley view*, Tech. Report UCB/EECS-2008-23, EECS Department, University of California, Berkeley, Mar 2008.

[ACU10] Thorsten Altenkirch, James Chapman, and Tarmo Uustalu, *Monads need not be endofunctors*, Foundations of Software Science and Computational Structures (2010), 297–311.

[ACU11] ———, *Relative monads formalised*, To appear in the Journal of Formalized Reasoning. Final version pending. (2011).

[ACU12] Danel Ahman, James Chapman, and Tarmo Uustalu, *When is a container a comonad?*, Foundations of Software Science and Computational Structures (2012), 74–88.

[Ada93] S. Adams, *Functional Pearls: Efficient sets–a balancing act*, Journal of functional programming **3** (1993), no. 4, 553–562.

[AFJW95] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge, *Multidimensional programming*, Oxford University Press, Oxford, UK, 1995.

[AM10] M. Aguiar and S.A. Mahajan, *Monoidal functors, species and Hopf algebras*, American Mathematical Society, 2010.

[App97] A.W. Appel, *Modern compiler implementation in ML*, Cambridge University Press, 1997.

[Atk09] R. Atkey, *Parameterised notions of computation*, Journal of Functional Programming **19** (2009), no. 3-4, 335–376.

[B+88] R. Bird et al., *Lectures on constructive functional programming*, Oxford University Computing Laboratory, Programming Research Group, 1988.

[Bac78] J. Backus, *Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs*, Communications of the ACM **21** (1978), no. 8, 613–641.

[Bar93] H. P. Barendregt, *Lambda calculi with types, Handbook of logic in computer science (vol. 2): background: computational structures*, 1993.

[BDM97] R.S. Bird and O. De Moor, *Algebra of programming,*, International series in computer science, vol. 5, Prentice Hall, 1997, p. 2.

[BdP96] G.M. Bierman and V.C.V. de Paiva, *Intuitionistic necessity revisited*, Proceedings of the Logic at Work Conference, Citeseer, 1996.

[BdP00] ———, *On an intuitionistic modal logic*, Studia Logica **65** (2000), no. 3, 383–416.

[Bek09] D. Bekki, *Monads and Meta-lambda Calculus*, New Frontiers in Artificial Intelligence (2009), 193–208.

[BG91] Stephen Brookes and Shai Geva, *Computational Comonads and Intensional Semantics*, Cambridge Univ. Press, 1991, pp. 1–44.

[Bir87] R.S. Bird, *An introduction to the theory of lists*, Proceedings of the NATO Advanced Study Institute on Logic of programming and calculi of discrete design, Springer-Verlag New York, Inc., 1987, pp. 5–42.

[BK00] A.J. Berrick and M.E. Keating, *Categories and modules with K-theory in view*, Cambridge studies in advanced mathematics, Cambridge University Press, 2000.

[BL69] R.M. Burstall and P.J. Landin, *Programs and their Proofs: an Algebraic Approach*, Machine intelligence **4** (1969), 17.

[BME+07] G. Booch, R. Maksimchuk, M. Engle, B. Young, J. Conallen, and K. Houston, *Object-oriented analysis and design with applications*, Addison-Wesley Professional, 2007.

[Bol11] Max Bolingbroke, *Constraint Kinds for GHC*, 2011, http://blog.omega-prime.co.uk/?p=127 (Retreived 14/09/11).

[BRP07] R.F. Barrett, P.C. Roth, and S.W. Poole, *Finite Difference Stencils Implemented Using Chapel*, Oak Ridge National Laboratory, Tech. Rep. ORNL Technical Report TM-2007/122 (2007).

[BS93] Stephen Brookes and Kathryn V Stone, *Monads and Comonads in Intensional Semantics*, Tech. report, Pittsburgh, PA, USA, 1993.

[BS11] J. Baez and M. Stay, *Physics, topology, logic and computation: a Rosetta Stone*, New structures for physics (2011), 95–172.

[Car47] Rudolf Carnap, *Meaning and Necessity*, enlarged edition, 1956. ed., University of Chicago Press, Chicago, 1947.

[CKJ05] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyton Jones, *Associated type synonyms*, ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ACM, 2005, pp. 241–253.

[CKJM05] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow, *Associated types with class*, SIGPLAN Not. **40** (2005), no. 1, 1–13.

[CKS07] Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan, *Finally Tagless, Partially Evaluated*, APLAS, 2007, pp. 222–238.

[CLWY06] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop, *Links: Web programming without tiers*, In 5th International Symposium on Formal Methods for Components and Objects (FMCO, Springer-Verlag, 2006.

[Col89] M.I. Cole, *Algorithmic skeletons: structured management of parallel computation*, Pitman, 1989.

[CR91] A. Carboni and R. Rosebrugh, *Lax monads: Indexed monoidal monads*, Journal of Pure and Applied Algebra **76** (1991), no. 1, 13–32.

[CU10] S. Capobianco and Tarmo Uustalu, *A Categorical Outlook on Cellular Automata*, vol. 13, 2010, pp. 88–99.

[CUV06] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene, *Recursive coalgebras from comonads*, Information and Computation **204** (2006), no. 4, 437–468.

[CWM99] Karl Crary, David Walker, and Greg Morrisett, *Typed memory management in a calculus of capabilities*, POPL '99, 1999.

[dB89] P. J. de Bruin., *Naturalness of polymorphism*, Tech. report, Technical Report CS 8916, Rijksuniversiteit Groningen, The Netherlands, 1989.

[DFH+93] J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While, *Parallel programming using skeleton functions*, PARLE'93 Parallel Architectures and Languages Europe, Springer, 1993, pp. 146–160.

[DHJG06] N.A. Danielsson, J. Hughes, P. Jansson, and J. Gibbons, *Fast and loose reasoning is morally correct*, POPL '06, ACM, 2006, pp. 206–217.

[DMV+08] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*, Proceedings of the 2008 ACM/IEEE conference on Supercomputing, IEEE Press, 2008, pp. 1–12.

[EK66] S. Eilenberg and G.M. Kelly, *Closed categories*, Springer, Berlin / New York, 1966.

[EM65] S. Eilenberg and J.C. Moore, *Adjoint functors and triples*, Illinois Journal of Mathematics **9** (1965), no. 3, 381–398.

[Fit09] Melvin Fitting, *Intensional Logic*, The Stanford Encyclopedia of Philosophy (Edward N. Zalta, ed.), spring 2009 ed., 2009.

[FM91] Maarten M. Fokkinga and Erik Meijer, *Program Calculation Properties of Continuous Algebras*, January 1991, Imported from EWI/DB PMS [db-utwente:tech:0000003528].

[FOK92] MM FOKKINGA, *Law and Order in Algorithmics*, PhD Thesis, University of Twente, Dept INF (1992).

[FQ03] Cormac Flanagan and Shaz Qadeer, *A type and effect system for atomicity*, Proceedings of Conference on Programming Language Design and Implementation, PLDI '03, 2003.

[Fre48] G. Frege, *Sense and reference*, The philosophical review **57** (1948), no. 3, 209–230.

[G+94] J. Gibbons et al., *An introduction to the Bird-Meertens formalism*, New Zealand Formal Program Development Colloquium Seminar, Hamilton, 1994, p. 102.

[Gam95] E. Gamma, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Professional, 1995.

[Gib02] J. Gibbons, *Calculating functional programs*, Algebraic and Coalgebraic Methods in the Mathematics of Program Construction (2002), 151–203.

[Gib03] ———, *Origami programming*, The Fun of Programming (J. Gibbons and O. de Moor, eds.), Cornerstones in Computing, Palgrave, 2003, pp. 148–203.

[GL86] David K. Gifford and John M. Lucassen, *Integrating functional and imperative programming*, Proceedings of Conference on LISP and func. prog., LFP '86, 1986.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J Scott, *Bounded linear logic: a modular approach to polynomial-time computability*, Theoretical computer science **97** (1992), no. 1, 1–66.

[GTL89] J.Y. Girard, P. Taylor, and Y. Lafont, *Proofs and types*, vol. 7, Cambridge University Press Cambridge, 1989.

[HCH08] R. Hirschfeld, P. Costanza, and M. Haupt, *An introduction to context-oriented programming with contexts*, Generative and Transformational Techniques in Software Engineering II (2008), 396–407.

[HH10] J. Havel and A. Herout, *Rendering Pipeline Modelled by Category Theory*, GraVisMa 2010 workshop proceedings, University of West Bohemia in Pilsen, 2010, pp. 101–105.

[HHM07] Russ Harmer, Martin Hyland, and Paul-Andre Mellies, *Categorical Combinatorics for Innocent Strategies*, LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (Washington, DC, USA), IEEE Computer Society, 2007, pp. 379–388.

[HHPJW96] C.V. Hall, K. Hammond, S. Peyton Jones, and P.L. Wadler, *Type classes in Haskell*, ACM Transactions on Programming Languages and Systems (TOPLAS) **18** (1996), no. 2, 109–138.

[HS08] J.R. Hindley and J.P. Seldin, *Lambda-calculus and combinators: an introduction*, Cambridge University Press, 2008.

[Hue97] G. Huet, *The zipper*, Journal of Functional Programming **7** (1997), no. 05, 549–554.

[Hug00] J. Hughes, *Generalising monads to arrows*, Science of computer programming **37** (2000), no. 1-3, 67–111.

[Hug04] ———, *Global variables in Haskell*, Journal of Functional Programming **14** (2004), no. 05, 489–502.

[Hug05] ———, *Programming with arrows*, Advanced Functional Programming (2005), 73–129.

[Jeu89] J. Jeuring, *Deriving algorithms on binary labelled trees*, Proceedings SION Computing Science in the Netherlands, PMG, 1989, pp. 229–249.

[JG89a] P. Jouvelot and D. K. Gifford, *Communication Effects for Message-Based Concurrency*, Tech. report, Massachusetts Institute of Technology, 1989.

[JG89b] P. Jouvelot and D.K. Gifford, *Reasoning about continuations with control effects*, vol. 24, ACM, 1989.

[JKS95] R. Jain, R. Kasturi, and B.G. Schunck, *Machine vision*, vol. 5, McGraw-Hill New York, 1995.

[Joh02] P.T. Johnstone, *Sketches of an elephant: a topos theory compendium*, vol. 2, Oxford University Press, USA, 2002.

[JR97] Bart Jacobs and Jan Rutten, *A tutorial on (Co)algebras and (Co)induction*, EATCS Bulletin **62** (1997), 62–222.

[JTH01] Simon Peyton Jones, Andrew. Tolmach, and Tony Hoare, *Playing by the rules: rewriting as a practical optimisation technique in GHC*, Haskell Workshop, vol. 1, 2001, pp. 203–233.

[KBB+07] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, *Effective Automatic Parallelization of Stencil Computations*, ACM Sigplan Notices **42** (2007), no. 6, 235–244.

[KCL+10] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier, *Regular, shape-polymorphic, parallel arrays in Haskell*, ICFP '10, ACM, 2010, pp. 261–272.

[KDW+06] S. Kamil, K. Datta, S. Williams, L. Oliker, J. Shalf, and K. Yelick, *Implicit and explicit optimizations for stencil computations*, Proceedings of the 2006 workshop on Memory system performance and correctness, ACM, 2006, pp. 51–60.

[Kel82] G.M. Kelly, *Basic concepts of enriched category theory*, vol. 64, Cambridge Univ Pr, 1982.

[Kie99] Richard B. Kieburtz, *Codata and Comonads in Haskell*, 1999.

[Kme12] Edward Kmett, *`Control.Comonad` package*, 2012, http://hackage.haskell.org/package/comonad, accessed September, 2012.

[Koc72] A. Kock, *Strong functors and monoidal monads*, Archiv der Mathematik **23** (1972), no. 1, 113–120.

[Lam80] J. Lambek, *From lambda-calculus to cartesian closed categories*, To HB Curry: essays on combinatory logic, lambda calculus and formalism (1980), 375–402.

[Lan66] Peter J. Landin, *The Next 700 Programming Languages*, Communications of the ACM **9** (1966), no. 3, 157–166.

[Les10] Michael Lesniak, *PASTHA: Parallelizing stencil calculations in Haskell*, Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming (New York, NY, USA), DAMP '10, ACM, 2010, pp. 5–14.

[LG88] J.M. Lucassen and D.K. Gifford, *Polymorphic effect systems*, Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1988, pp. 47–57.

[LK11] B. Lippmeier and G. Keller, *Efficient parallel stencil convolution in Haskell*, ACM SIGPLAN Notices, vol. 46, ACM, 2011, pp. 59–70.

[LLMS00] J.R. Lewis, J. Launchbury, E. Meijer, and M.B. Shields, *Implicit parameters: Dynamic scoping with static types*, Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 2000, p. 118.

[LS88] J. Lambek and P.J. Scott, *Introduction to higher order categorical logic*, Cambridge Univ Pr, 1988.

[Mai07] Geoffrey Mainland, *Why it's Nice to be quoted: Quasiquoting for Haskell*, Haskell '07 (New York, NY, USA), ACM, 2007, pp. 73–82.

[Mal90] G. Malcolm, *Data structures and program transformation*, Science of computer programming **14** (1990), no. 2, 255–279.

[McB01] C. McBride, *The derivative of a regular type is its type of one-hole contexts*, Unpublished manuscript (2001).

[McB11] ———, *Functional pearl: Kleisli arrows of outrageous fortune*, Journal of Functional Programming (to appear) (2011).

[MCH08] Tom Murphy, VII., Karl Crary, and Robert Harper, *Type-safe distributed programming with ML5*, TGC'07, 2008, pp. 108–123.

[Mee86] L. Meertens, *Algorithmics – towards programming as a mathematical activity*, CWI Monographs (JW de Bakker, M. Hazewinkel, JK Lenstra, eds.) North Holland, Puhl. Co **1** (1986).

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson, *Functional programming with bananas, lenses, envelopes and barbed wire*, Functional Programming Languages and Computer Architecture, Springer, 1991, pp. 124–144.

[Mil78] R. Milner, *A theory of type polymorphism in programming*, Journal of computer and system sciences **17** (1978), no. 3, 348–375.

[ML98] S. Mac Lane, *Categories for the working mathematician*, Springer verlag, 1998.

[MLS84] P. Martin-Löf and G. Sambin, *Intuitionistic type theory*, vol. 17, Bibliopolis Naples, Italy, 1984.

[Mog89] E. Moggi, *Computational lambda-calculus and monads*, Logic in Computer Science, 1989. LICS'89, Proceedings., Fourth Annual Symposium on, IEEE, 1989, pp. 14–23.

[Mog91] ———, *Notions of computation and monads*, Inf. Comput. **93** (1991), no. 1, 55–92.

[Mon70] R. Montague, *Universal grammar*, Theoria **36** (1970), no. 3, 373–398.

[MP08] C. McBride and R. Paterson, *Functional pearl: Applicative programming with effects*, Journal of functional programming **18** (2008), no. 1, 1–13.

[MVCHP04] Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning, *A Symmetric Modal Lambda Calculus for Distributed Computing*, Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science, LICS '04, 2004, pp. 286–295.

[Myc80] Alan Mycroft, *The Theory and Practice of Transforming Call-by-need into Call-by-value*, Symposium on Programming, 1980, pp. 269–281.

[NN99] Flemming Nielson and Hanne Nielson, *Type and effect systems*, Correct System Design (1999), 114–136.

[NPP08] A. Nanevski, F. Pfenning, and B. Pientka, *Contextual modal type theory*, ACM Transactions on Computational Logic (TOCL) **9** (2008), no. 3, 23.

[OBM10] Dominic Orchard, Max Bolingbroke, and Alan Mycroft, *Ypnos: declarative, parallel structured grid programming*, DAMP '10: Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming (New York, NY, USA), ACM, 2010, pp. 15–24.

[OED06] *Concise Oxford English Dictionary*, 2006.

[OM11] Dominic Orchard and Alan Mycroft, *Efficient and Correct Stencil Computation via Pattern Matching and Static Typing*, Electronic Proceedings in Theoretical Computer Science **66** (2011), 68–92.

[OM12a] ———, *Mathematically structures for data types with restricted parametricity*, TFP '12: Trends in Functional Programming, Pre-proceedings, 2012.

[OM12b] ———, *A Notation for Comonads*, IFL '12: Implementation and Application of Functional Languages, Revised Selected Papers, vol. 8241, 2012.

[Orc11] Dominic Orchard, *The Four Rs of Programming Language Design*, Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software, ACM, 2011, pp. 157–162.

[Orc12] ———, *When monads and comonads overlap*, Tech. report, 2012.

[OS10] Dominic Orchard and Tom Schrijvers, *Haskell type constraints unleashed*, Functional and Logic Programming, vol. 6009/2010, Springer Berlin / Heidelberg, April 2010, pp. 56–71.

[Pap03] C.H. Papadimitriou, *Computational complexity*, John Wiley and Sons Ltd., 2003.

[Par00] A. Pardo, *Towards merging recursion and comonads*, Proceedings of the 2nd Workshop on Generic Programming, WGP00, Citeseer, 2000, pp. 50–68.

[Pat01] R. Paterson, *A new notation for arrows*, ACM SIGPLAN Notices, vol. 36, ACM, 2001, pp. 229–240.

[PD01] F. Pfenning and R. Davies, *A judgmental reconstruction of modal logic*, Mathematical structures in computer science **11** (2001), no. 4, 511–540.

[Pie02] B.C. Pierce, *Types and programming languages*, MIT press, 2002.

[Pie05] ———, *Advanced topics in types and programming languages*, The MIT Press, 2005.

[Pip09] Dan Piponi, *Evaluating cellular automata is comonadic*, December 2006, Last retreived September 2009, http://blog.sigfpe.com/2006/12/evaluating-cellular-automata-is.html.

[PJ+03] Simon Peyton Jones et al., *Haskell 98 Language and Libraries: The Revised Report*, Cambridge University Press, May 2003.

[PJVWW06] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn, *Simple unification-based type inference for GADTs*, Proceedings of ACM SIGPLAN international conference on Functional programming (New York, NY, USA), ICFP '06, ACM, 2006, pp. 50–61.

[PJWW04] Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich, *Wobbly Types: Type Inference for Generalised Algebraic Data Types*, Tech. report, July 2004.

[PMS11] Tomas Petricek, Alan Mycroft, and Don Syme, *Extending monads with pattern matching*, ACM SIGPLAN Notices, vol. 46, ACM, 2011, pp. 1–12.

[POM13] Tomas Petricek, Dominic A. Orchard, and Alan Mycroft, *Coeffects: Unified Static Analysis of Context-Dependence*, ICALP (2), 2013, pp. 385–397.

[PP95] J. Plaice and J. Paquet, *Introduction to intensional programming*, 1995.

[PR97] John Power and Edmund Robinson, *Premonoidal categories and notions of computation*, Mathematical structures in computer science **7** (1997), no. 5, 453–468.

[Pra65] D. Prawitz, *Natural deduction: A proof-theoretical study*, no. 3, Almqvist & Wiksell Stockholm, 1965.

[PS11] Tomas Petricek and Don Syme, *Joinads: a retargetable control-flow construct for reactive, parallel and concurrent programming*, Practical Aspects of Declarative Languages (2011), 205–219.

[PS12] ———, *Syntax Matters: writing abstract computations in F#*, Pre-proceedings of TFP, 2012.

[PT99] John Power and Hayo Thielecke, *Closed freyd-and κ-categories*, Automata, Languages and Programming, Springer, 1999, pp. 625–634.

[PW95] F. Pfenning and H.C. Wong, *On a Modal [lambda]-Calculus for S41*, Electronic Notes in Theoretical Computer Science **1** (1995), 515–534.

[PW02] John Power and Hiroshi Watanabe, *Combining a monad and a comonad*, Theoretical Computer Science **280** (2002), no. 1-2, 137–162.

[RP93] J.C. Reynolds and G.D. Plotkin, *On functors expressible in the polymorphic typed lambda calculus*, Information and Computation **105** (1993), no. 1, 1–29.

[SAW94] B. Schilit, N. Adams, and R. Want, *Context-aware computing applications*, Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on, IEEE, 1994, pp. 85–90.

[SC10] Thierry Sans and Iliano Cervesato, *QWeSST for Type-Safe Web Programming*, Third International Workshop on Logics, Agents, and Mobility (Edinburgh, Scotland, UK), LAM'10, 2010.

[Sch03] Sven-Bodo Scholz, *Single Assignment C: efficient support for high-level array operations in a functional setting*, Journal of Functional Programming **13** (2003), no. 6, 1005–1059.

[Sha02] C. Shan, *Monads for natural language semantics*, Arxiv preprint cs/0205026 (2002).

[Sim94] A.K. Simpson, *The proof theory and semantics of intuitionistic modal logic*.

[Sim95] Alex K Simpson, *Categorical completeness results for the simply-typed lambda-calculus*, Typed lambda calculi and applications, Springer, 1995, pp. 414–427.

[SJCS08] T. Schrijvers, S. Peyton Jones, M. Chakravarty, and M. Sulzmann, *Type checking with open type functions*, SIGPLAN Not. **43** (2008), no. 9, 51–62.

[SM03] Andrei Sabelfeld and Andrew C. Myers, *Language-based Information-Flow Security*, IEEE Journal on Selected Areas in Communications **21** (2003), no. 1.

[Smi06] C. Smith, *On vertex-vertex systems and their use in geometric and biological modelling*, University of Calgary, 2006.

[SP00] Alex Simpson and Gordon Plotkin, *Complete axioms for categorical fixed-point operators*, Logic in Computer Science, 2000. Proceedings. 15th Annual IEEE Symposium on, IEEE, 2000, pp. 30–41.

[Spi89] M. Spivey, *A categorical approach to the theory of lists*, Mathematics of Program Construction, Springer, 1989, pp. 399–408.

[Ste09] Don Stewart, *Domain Specific Languages for Domain Specific Problems*, Workshop on Non-Traditional Programming Models for High-Performance Computing, LACSS, 2009.

[SZD+98] P. Schröder, D. Zorin, T. DeRose, DR Forsey, L. Kobbelt, M. Lounsbery, and J. Peters, *Subdivision for modeling and animation*, ACM SIGGRAPH Course Notes **12** (1998).

[TJ92] J.P. Talpin and P. Jouvelot, *The type and effect discipline*, Logic in Computer Science, 1992. LICS'92., Proceedings of the Seventh Annual IEEE Symposium on, IEEE, 1992, pp. 162–173.

[UV02] Tarmo Uustalu and Varmo Vene, *The dual of substitution is redecoration*, Trends in functional programming **3** (2002), 99–110.

[UV05] ———, *Signals and Comonads*, no. 7, 1311–1326, Precedes the *"The Essence of Dataflow"* but contains much of the same material.

[UV06] ———, *The Essence of Dataflow Programming*, Lecture Notes in Computer Science **4164** (November 2006), 135–167.

[UV07] ———, *Comonadic functional attribute evaluation*, Trends in Functional Programming **6** (2007), 145–162.

[UV08] ———, *Comonadic Notions of Computation*, Electron. Notes Theor. Comput. Sci. **203** (2008), no. 5, 263–284.

[UV11] ———, *The recursion scheme from the cofree recursive comonad*, Electronic Notes in Theoretical Computer Science **229** (2011), no. 5, 135–157.

[UVP01] Tarmo Uustalu, Varmo Vene, and Alberto Pardo, *Recursion schemes from comonads*, Nordic Journal of Computing **8** (2001), no. 3, 366–390.

[Ven00] Varmo Vene, *Categorical programming with inductive and coinductive types*, Tartu University Press, 2000.

[Ver92] D. Verity, *Enriched categories, internal categories and change of base*, no. 20, University of Cambridge, 1992.

[vFH08] K. von Fintel and I. Heim, *Intensional semantics*, 2008, http://web.mit.edu/linguistics/people/faculty/heim/papers/fintel-heim-intensional.pdf.

[VIS96] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith, *A sound type system for secure flow analysis*, J. Comput. Secur. **4** (1996), 167–187.

[WA85] William W. Wadge and Edward A. Ashcroft, *LUCID, the dataflow programming language*, Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[Wad88] P. Wadler, *Strictness analysis aids time analysis*, Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1988, pp. 119–132.

[Wad89] ———, *Theorems for free!*, Proceedings of the fourth international conference on Functional programming languages and computer architecture, ACM, 1989, pp. 347–359.

[Wad90a] ———, *Deforestation: Transforming programs to eliminate trees*, Theoretical computer science **73** (1990), no. 2, 231–248.

[Wad90b] Philip Wadler, *Linear Types Can Change the World!*, Programming Concepts and Methods, North, 1990.

[Wad92a] P. Wadler, *The essence of functional programming*, Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1992, pp. 1–14.

[Wad92b] Philip Wadler, *Comprehending Monads*, Mathematical Structures in Computer Science, 1992, pp. 61–78.

[Wad95a] William Wadge, *Monads and Intensionality*, International Symposium on Lucid and Intensional Programming '95, 1995.

[Wad95b] P. Wadler, *Monads for functional programming*, Advanced Functional Programming (1995), 24–52.

[WB89] P. Wadler and S. Blott, *How to make ad-hoc polymorphism less ad hoc*, Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM, 1989, pp. 60–76.

[WR12] A.N. Whitehead and B. Russell, *Principia mathematica*, vol. 2, University Press, 1912.

[WT03] Philip Wadler and Peter Thiemann, *The marriage of effects and monads*, ACM Trans. Comput. Logic **4** (2003), 1–32.

[YHG+07] K. Yelick, P. Hilfinger, S. Graham, D. Bonachea, J. Su, A. Kamil, K. Datta, P. Colella, and T. Wen, *Parallel Languages and Compilers: Perspective from the Titanium experience*, International Journal of High Performance Computing Applications **21** (2007), no. 3, 266.

Appendix A

Category theory basics

A.1. Categories

Central to category theory is the concept of a *morphism* describing a *mapping* between two structures or entities. Morphisms have a *source* object and *target* object, written $X \to Y$ for source X and target Y. In categorical logic and semantics, morphisms characterise both programs, mapping inputs to outputs, and implications from one proposition to another.

Definition A.1.1. A *category* $\mathbb{C}$ comprises a class[1] of *objects* $\mathbb{C}_0$, a class of *morphisms* $\mathbb{C}_1$, a (partial) binary operation $\circ$ for the *composition* of morphisms where for all $f : A \to B, g : B \to C \in \mathbb{C}_1$ then $g \circ f : A \to C \in \mathbb{C}_1$, and for every object $A \in \mathbb{C}_0$ an *identity* morphism $id_A : A \to A \in \mathbb{C}_0$, satisfying three axioms:

(*associativity*)	$h \circ (g \circ f) = (h \circ g) \circ f$	(for all $f : A \to B, g : B \to C, h : C \to D \in \mathbb{C}_1$)
(*right unit*)	$f \circ id_A = f$	(for all $f : A \to B \in \mathbb{C}_1$)
(*left unit*)	$id_B \circ f = f$	(for all $f : A \to B \in \mathbb{C}_1$)

Thus, categories are monoids where the totality property is relaxed since not all morphisms compose, only those that agree in the corresponding target and source objects.

Definition A.1.2. For a category $\mathbb{C}$, the collection of morphisms between two objects $A, B \in \mathbb{C}_0$ is called the *hom-set* (although it may not be a set) denoted $\mathbb{C}(A, B)$.

Example A.1.3. The category **Set**, has sets as objects and functions, in the set-theoretical sense, as morphisms. Function composition provides composition in the category and identity functions provide the identity morphisms.

Since the set of partial operations is a superset of total operations, *monoids* are thus *categories*:

Example A.1.4. A monoid $(X, \otimes, 1)$ has a corresponding category with a single object $\bullet$, for every $x \in X$ a morphism $x : \bullet \to \bullet$, and the identity morphism $id_\bullet : \bullet \to \bullet$ for the unit 1.

Example A.1.5. Given a partially-ordered set $(X, \leqslant)$ there is a category modelling the partial order $\mathbb{X}$, where $\mathbb{X}_0 = X$ and for all $A, B \in X$ such that $A \leqslant B$, then $f : A \to B \in \mathbb{X}_1$. Identities are provided by the reflexivity of a partial order, and composition by its transitivity.

Definition A.1.6. For a category $\mathbb{C}$, the *dual* category $\mathbb{C}^{\mathsf{op}}$ is has objects $\mathbb{C}_0^{\mathsf{op}} = \mathbb{C}_0$ and morphisms $\mathbb{C}^{\mathsf{op}}(Y, X) = \mathbb{C}(X, Y)$ *i.e.* the direction of arrows, and therefore composition, is reversed.

A.2. Functors

Functors. Products are an example of a structure over the objects of a category (*i.e.* for all $A, B \in \mathbb{C}_0$ then $A \times B \in \mathbb{C}_0$) which has an accompanying notion of lifting morphisms to

[1] A *class* is a collection of objects which is possibly, but not necessarily, a set. The terminology of *classes* is used to avoid size paradoxes *i.e.* the *set of all sets* is replaced by the *class of all sets*.

operate on the components of the structure (alternatively a notion of the structure applied to morphisms): for all $f : A \to B, g : C \to D \in \mathbb{C}_1$ then $f \times g : A \times C \to B \times D \in \mathbb{C}_1$, is defined:

$$f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle : A \times C \to B \times D \in \mathbb{C}_1 \tag{68}$$

An object mapping coupled with a morphism mapping is a *functor* if certain axioms are satisfied.

Definition A.2.1. A *functor* F is a mapping between two categories $\mathbb{C}, \mathbb{D}$ written $\mathsf{F} : \mathbb{C} \to \mathbb{D}$, with an *object mapping* from all objects $A \in \mathbb{C}_0$ to $\mathsf{F}A \in \mathbb{D}_0$ and a *morphism mapping* from all morphisms $f : A \to B \in \mathbb{C}_1$ to $\mathsf{F}f : \mathsf{F}A \to \mathsf{F}B \in \mathbb{D}_1$, satisfying:

[F1] $\mathsf{F}(f \circ g) = \mathsf{F}f \circ \mathsf{F}g$ (for all $f : A \to B, g : B \to C \in \mathbb{C}_1$)
[F2] $\mathsf{F}id_A = id_{\mathsf{F}A}$ (for all $A \in \mathbb{C}_0$)

Thus, by [F1-2], functors preserve the composition structure of morphisms from $\mathbb{C}$ to $\mathbb{D}$.

Example A.2.2. A category $\mathbb{C}$ with products has a functor $(- \times -) : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ mapping from the *product category*[2] $\mathbb{C} \times \mathbb{C}$, with:

- object mapping $(A, B) \in (\mathbb{C} \times \mathbb{C})_0$ to $A \times B \in \mathbb{C}_0$
- morphism mapping $(f : A \to B, g : C \to D) \in (\mathbb{C} \times \mathbb{C})_1$ to $f \times g : A \times C \to B \times D : \mathbb{C}_1$ as defined in (68).

The functor laws follow from the definition of the morphism mapping (68) and the product axioms. A functor mapping from a product category is sometimes called a *bifunctor*.

Example A.2.3. Hom-sets for a category can be defined by a bifunctor called the *hom*-functor (or *external hom*) $\mathbb{C}(-,-) : \mathbb{C}^{\mathsf{op}} \times \mathbb{C} \to \mathbb{D}$ (where $\mathbb{D} = \mathbf{Set}$ if hom-sets are actually sets):

- $\mathbb{C}(A, B)$ is the object in $\mathbb{D}$ representing the morphisms from A to B in $\mathbb{C}$.
- $\mathbb{C}(f : A \to B, g : X \to Y) = \ \mathbb{C}(B, X) \xrightarrow{g \circ - \circ f} \mathbb{C}(A, Y)$

Example A.2.4. A product category $(\mathbb{C} \times \mathbb{D})$ has *projection functors* $\mathsf{Proj}_1 : \mathbb{C} \times \mathbb{D} \to \mathbb{C}$ and $\mathsf{Proj}_2 : \mathbb{C} \times \mathbb{D} \to \mathbb{D}$ where:

$$\mathsf{Proj}_1\,(A, B) = A \qquad \mathsf{Proj}_2\,(A, B) = B$$
$$\mathsf{Proj}_1\,(f, g) = f \qquad \mathsf{Proj}_2\,(f, g) = g$$

Definition A.2.5. An *endo*functor has equal source and target categories *i.e.* $\mathsf{F} : \mathbb{C} \to \mathbb{C}$.

Example A.2.6. The identity functor for a category $\mathbb{C}$ is an endofunctor $1_{\mathbb{C}} : \mathbb{C} \to \mathbb{C}$ with identity mappings for objects, $1_{\mathbb{C}}A = A$ for all $A \in \mathbb{C}_0$, and morphisms, $1_{\mathbb{C}}f = f$ for all $f : A \to B \in \mathbb{C}_1$.

Remark A.2.7. The category of categories **Cat** has categories as objects and functors as morphisms with identity functors for each category object as the identity morphisms and composition of functors defined simply as the composite $(\mathsf{G} \circ \mathsf{F})A = \mathsf{G}(\mathsf{F}A)$ for objects and $(\mathsf{G} \circ \mathsf{F})f = \mathsf{G}(\mathsf{F}f)$

[2] A product category $\mathbb{C} \times \mathbb{D}$ extends the notion of a Cartesian-product of two sets to categories, with objects (A, B) and morphisms (f, g) for all possible pairings of objects $A \in \mathbb{C}_0, B \in \mathbb{D}_0$ and morphisms $f \in \mathbb{C}_1, g \in \mathbb{D}_1$, where composition is defined component-wise *i.e.* $(g_1, g_2) \circ (f_1, f_2) = (g_1 \circ f_1, g_2 \circ f_2)$.

for morphisms (checking the functor laws is straightforward) and from now on will be written by juxtaposition *i.e.* $\mathsf{GF} = \mathsf{G} \circ \mathsf{F}$.

The notion of a functor thus captures notions of parameterised data types where, under the types as objects interpretation, the object-mapping of a functor instantiates a data type and the morphism-mapping provides a *map*-like operation over the type. This interpretation of functors is is key to their use in categorical programming (see Section 2.4). For the categorical semantics, the functorial nature of products means that operations on contexts can be arbitrarily nested.

A.3. Natural transformations

Natural transformations are essentially morphisms between functors which described operations which are generic in their objects which parameterise the functors.

Definition A.3.1. Given two functors $\mathsf{F}, \mathsf{G} : \mathbb{C} \to \mathbb{D}$, a *natural transformation* $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ is a family of morphisms $\alpha_X : \mathsf{F}X \to \mathsf{G}X \in \mathbb{D}_1$ indexed by objects $X \in \mathbb{C}_0$, such that, for all morphisms $f : X \to Y \in \mathbb{C}_1$, α satisfies the *naturality property*:

$$\begin{array}{ccc} \mathsf{F}(X) & \xrightarrow{\alpha_x} & \mathsf{G}(X) \\ {\scriptstyle \mathsf{F}f}\downarrow & & \downarrow{\scriptstyle \mathsf{G}f} \\ \mathsf{F}(Y) & \xrightarrow[\alpha_y]{} & \mathsf{G}(Y) \end{array} \qquad \text{or written equationally:} \quad \mathsf{G}f \circ \alpha_x = \alpha_y \circ \mathsf{F}f \tag{69}$$

Natural transformations will be written equivalently as either $\alpha : \mathsf{F} \dot{\to} \mathsf{G}$ or $\alpha_A : \mathsf{F}A \to \mathsf{G}A$.

Example A.3.2. The projection maps of products can be described by natural transformations $\pi_1 : (- \times -) \dot{\to} \mathsf{Proj}_1$ and $\pi_2 : (- \times -) \dot{\to} \mathsf{Proj}_2$, or for clarity they may be denoted $\pi_{1_{A,B}} : A \times B \to A$ and $\pi_{2_{A,B}} : A \times B \to B$.

Example A.3.3. The *exchange operation* $\chi_{A,B,C} : (A \times B) \times C \to (A \times C) \times B$ in Section 2.2.2 is a natural transformation defined:

$$\chi_{A,B,C} = \langle \pi_{1_{A,B}} \times id_C, \pi_{2_{A,B}} \circ \pi_{1_{(A \times B),C}} \rangle \tag{70}$$

where $(\pi_{1_{A,B}} \times id_C) : (A \times B) \times C \to (A \times C)$ and $\pi_{2_{A,B}} \circ \pi_{1_{(A \times B),C}} : (A \times B) \times C \to B$.

Appendix B

Additional definitions

The following provides full definitions and proofs omitted from the body of the dissertation.

B.1. Background

B.1.1. Reductions for the simply-typed λ-calculus

The *full-β* scheme allows ubiquitous term reduction:

$$(\zeta_1)\frac{e_1 \rightsquigarrow e_1'}{e_1\, e_2 \rightsquigarrow e_1'\, e_2} \qquad (\zeta_2)\frac{e_2 \rightsquigarrow e_2'}{e_1\, e_2' \rightsquigarrow e_1\, e_2'} \qquad (\zeta_3)\frac{e \rightsquigarrow e'}{\lambda x.e \rightsquigarrow \lambda x.e'}$$

Since λ^τ is *pure*, the full-β scheme, with (β) and (η), provides a confluent and terminating rewrite system [**HS08**, Appendix 2]. For particular specialisations or extensions to λ^τ, a different reduction strategy may be required, such as *call-by-value* or *call-by-name* evaluation:

- Call-by-name (CBN) evaluation has just the (β) and (ζ_1) rules.
- Call-by-value (CBV) evaluation requires a distinction between *values* and *computations*. The λ-calculus syntax can be divided into value t_{V} and not-necessarily-value t terms:

  $$t := t_1\, t_2 \mid t_{\mathrm{V}} \qquad t_{\mathrm{V}} := \lambda v.t \mid v$$

  CBV then has (ζ_2), a specialised (β) rule for substituted values (not general expressions) into function bodies, and a specialised (ζ_1) rule where the right-hand term must be a value:

  $$(\beta\text{-CBV}) \quad (\lambda x.t_1)\, t_{\mathrm{V}} \rightsquigarrow t_1[x := t_{\mathrm{V}}] \qquad\qquad (\zeta_1\text{-CBV})\frac{t_1 \rightsquigarrow t_1'}{t_1\, t_{\mathrm{V}} \rightsquigarrow t_1'\, t_{\mathrm{V}}}$$

B.1.2. Monoidal categories

Definition B.1.1. [**ML98**] A *monoidal category* $(\mathbb{C}, \otimes, I)$ comprises a category $\mathbb{C}$ along with a *bifunctor* $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$, a unit object $I \in \mathbb{C}_0$, and three natural isomorphisms:

$$\begin{array}{ll} (\textit{associativity}) & \alpha_{A,B,C} : (A \otimes B) \otimes C \cong A \otimes (B \otimes C) \\ (\textit{left unit}) & \lambda_A : (I \otimes A) \cong A \\ (\textit{right unit}) & \rho_A : (A \otimes I) \cong A \end{array}$$

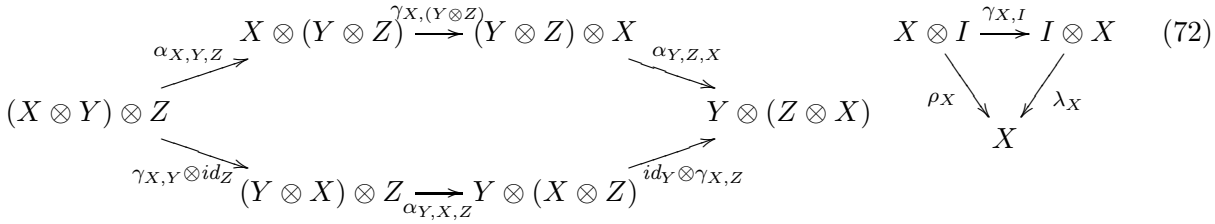
such that the following diagrams commute:

$$\begin{array}{ccc} ((X \otimes Y) \otimes Z) \otimes W & \xrightarrow{\alpha_{X,Y,Z} \otimes id_W} & (X \otimes (Y \otimes Z)) \otimes W \\ \downarrow{\scriptstyle \alpha_{(X\otimes Y),Z,W}} & & \downarrow{\scriptstyle \alpha_{X,(Y\otimes Z),W}} \\ (X \otimes Y) \otimes (Z \otimes W) & & X \otimes ((Y \otimes Z) \otimes W) \\ & \searrow^{\alpha_{X,Y,(Z\otimes W)}} & \downarrow{\scriptstyle id_X \otimes \alpha_{Y,Z,W}} \\ & & X \otimes (Y \otimes (Z \otimes W)) \end{array} \qquad \begin{array}{ccc} (X \otimes I) \otimes Y & \xrightarrow{\alpha_{X,I,Y}} & X \otimes (I \otimes Y) \\ & {\scriptstyle \rho_X \otimes id_Y} \searrow \quad \swarrow {\scriptstyle id_X \otimes \lambda_Y} & \\ & X \otimes Y & \end{array} \tag{71}$$

Definition B.1.2. A monoidal category $(\mathbb{C}, \otimes, I)$ is *symmetric* if it has the natural isomorphism:

$$(\textit{symmetry}) \quad \gamma_{A,B} : (A \otimes B) \cong (B \otimes A)$$

such that the following diagrams commute:



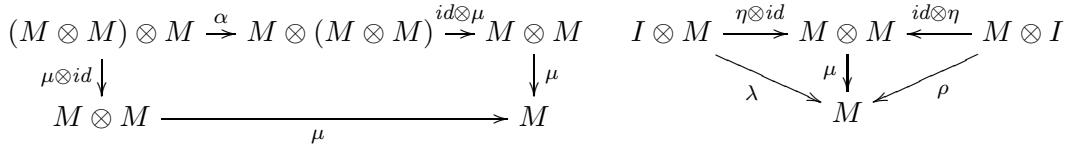
(72)

Example B.1.3. A category $\mathbb{C}$ with products $\times$ and terminal object 1 is symmetric monoidal, $(\mathbb{C}, \times, 1)$, with the associativity, unit, and symmetry isomorphisms defined

$$\begin{array}{lll}
\alpha_{A,B,C} & = \langle \pi_1 \circ \pi_1, \pi_2 \times id_C \rangle & : (A \times B) \times C \to A \times (B \times C) \\
\alpha^{-1}_{A,B,C} & = \langle id_A \times \pi_1, \pi_2 \circ \pi_2 \rangle & : A \times (B \times C) \to (A \times B) \times C \\
\lambda_A & = \pi_2 & : 1 \times A \to A \\
\lambda^{-1}_A & = \langle !_A, id_A \rangle & : A \to 1 \times A \\
\rho_A & = \pi_1 & : A \times 1 \to A \\
\rho^{-1}_A & = \langle id_A, !_A \rangle & : A \to A \times 1 \\
\gamma_{A,B} & = \langle \pi_2, \pi_1 \rangle = \gamma^{-1} & : A \times B \to B \times A
\end{array} \tag{73}$$

The coherence conditions for the monoidal category and the isomorphism properties follow straightforwardly from the universal properties of products and terminal morphisms.

Definition B.1.4. Given a monoidal category $(\mathbb{C}, \otimes, I)$, a *monoid* comprises an object M with two morphisms $\mu : M \otimes M \to M$ and $\eta : I \to M$ called the *multiplication* and *unit* respectively, satisfying the following properties:



Definition B.1.5. [**AM10**][Chapter 3] (first seen in Section 3.3, Definition 3.3.1) Given a functor $\mathsf{F} : \mathbb{C} \to \mathbb{D}$ between two monoidal categories $(\mathbb{C}, \otimes, I_{\mathbb{C}})$ and $(\mathbb{D}, \bullet, I_{\mathbb{D}})$, a *lax monoidal functor* has the following natural transformation and morphism [**EK66**]:

$$\mathsf{m}_{A,B} : \mathsf{F}A \bullet \mathsf{F}B \to \mathsf{F}(A \otimes B)$$
$$\mathsf{m}_1 : I_{\mathbb{D}} \to \mathsf{F}I_{\mathbb{C}}$$

satisfying a number of coherence conditions that $\mathsf{m}_{A,B}$ and m_1 preserve the associativity and unit natural transformations for $\mathbb{C}$ and $\mathbb{D}$:

$$\begin{array}{ccc}
(\mathsf{F}X \bullet \mathsf{F}Y) \bullet \mathsf{F}Z & \xrightarrow{\alpha_{\mathbb{D}X,Y,Z}} & \mathsf{F}X \bullet (\mathsf{F}Y \bullet \mathsf{F}Z) \\
\downarrow{\scriptstyle \mathsf{m}_{X,Y} \bullet id_{\mathsf{F}Z}} & & \downarrow{\scriptstyle id_{\mathsf{F}X} \bullet \mathsf{m}_{Y,Z}} \\
\mathsf{F}(X \otimes Y) \bullet \mathsf{F}Z & & \mathsf{F}X \bullet \mathsf{F}(Y \otimes Z) \\
\downarrow{\scriptstyle \mathsf{m}_{X \otimes Y,Z}} & & \downarrow{\scriptstyle \mathsf{m}_{X,Y \otimes Z}} \\
\mathsf{F}((X \otimes Y) \otimes Z) & \xrightarrow[\mathsf{F}\alpha_{\mathbb{C}X,Y,Z}]{} & \mathsf{F}(X \otimes (Y \otimes Z))
\end{array}
\qquad
\begin{array}{ccc}
\mathsf{F}X \bullet I_{\mathbb{D}} & \xrightarrow{id_{\mathsf{F}X} \bullet \mathsf{m}_1} & \mathsf{F}X \bullet \mathsf{F}I_{\mathbb{C}} \\
\downarrow{\scriptstyle \rho_{\mathbb{D}}} & & \downarrow{\scriptstyle \mathsf{m}_{X,I_{\mathbb{C}}}} \\
\mathsf{F}X & \xleftarrow[\mathsf{F}\rho_{\mathbb{C}}]{} & \mathsf{F}(X \otimes I_{\mathbb{C}})
\end{array}
\qquad
\begin{array}{ccc}
I_{\mathbb{D}} \bullet \mathsf{F}X & \xrightarrow{\mathsf{m}_1 \bullet id_{\mathsf{F}X}} & \mathsf{F}I_{\mathbb{C}} \bullet \mathsf{F}X \\
\downarrow{\scriptstyle \lambda_{\mathbb{D}}} & & \downarrow{\scriptstyle \mathsf{m}_{I_{\mathbb{C}},X}} \\
\mathsf{F}X & \xleftarrow[\mathsf{F}\lambda_{\mathbb{C}}]{} & \mathsf{F}(I_{\mathbb{C}} \otimes X)
\end{array}
\tag{74}$$

Thus $\mathsf{m}_{A,B}$ and m_1 define a homomorphism (under F) between the two monoidal structures of $\mathbb{C}$ and $\mathbb{D}$. There are number of variations of monoidal functors:

- A *colax monoidal functor* (sometimes *lax comonoidal* or *oplax monoidal*) has the dual operations, $\mathsf{n}_{A,B} : \mathsf{F}(A \otimes B) \to \mathsf{F}A \bullet \mathsf{F}B$ and $\mathsf{n}_1 : \mathsf{F}I_{\mathbb{C}} \to I_{\mathbb{D}}$ where the above diagrams have the direction of the m replaced with the reverse n arrows.
- A *strong monoidal functor* is a monoidal functor where $\mathsf{m}_{A,B}$ and m_1 are isomorphisms, thus a strong monoidal functor is both lax and colax, where $\mathsf{m}_{A,B}^{-1} = \mathsf{n}_{A,B}$ and $\mathsf{m}_1^{-1} = \mathsf{n}_1^{-1}$.
- A *strict monoidal functor* is a monoidal functor where $\mathsf{m}_{A,B}$ and m_1 are identities.

Lemma B.1.6. *If $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ is a strong monoidal endofunctor where $\mathbb{C}$ has categorical products, then the following diagram commutes:*

$$\begin{array}{ccccc}
 & & \mathsf{F}A \times \mathsf{F}B & & \\
 & \swarrow^{\pi_1} & \downarrow{\scriptstyle \mathsf{m}_{A,B}} & \searrow^{\pi_2} & \\
\mathsf{F}A & \xleftarrow[\mathsf{F}\pi_1]{} & \mathsf{F}(A \times B) & \xrightarrow[\mathsf{F}\pi_2]{} & \mathsf{F}B
\end{array}
\tag{75}$$

Proof. Since $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ is strong monoidal, the *colax* conditions hold:

$$\begin{array}{ccc}
\mathsf{F}X \bullet I_{\mathbb{D}} & \xleftarrow{id_{\mathsf{F}X} \bullet \mathsf{n}_1} & \mathsf{F}X \bullet \mathsf{F}I_{\mathbb{C}} \\
\downarrow{\scriptstyle \lambda_{\mathbb{D}}} & & \uparrow{\scriptstyle \mathsf{n}_{X,I_{\mathbb{C}}}} \\
\mathsf{F}X & \xleftarrow[\mathsf{F}\lambda_{\mathbb{C}}]{} & \mathsf{F}(X \otimes I_{\mathbb{C}})
\end{array}
\qquad
\begin{array}{ccc}
I_{\mathbb{D}} \bullet \mathsf{F}X & \xleftarrow{\mathsf{n}_1 \bullet id_{\mathsf{F}X}} & \mathsf{F}I_{\mathbb{C}} \bullet \mathsf{F}X \\
\downarrow{\scriptstyle \rho_{\mathbb{D}}} & & \uparrow{\scriptstyle \mathsf{n}_{I_{\mathbb{C}},X}} \\
\mathsf{F}X & \xleftarrow[\mathsf{F}\rho_{\mathbb{C}}]{} & \mathsf{F}(I_{\mathbb{C}} \otimes X)
\end{array}$$

(the dual of the bottom two squares in (74)). Since the monoidal structure is $(\mathbb{C}, \times, 1)$ (*i.e.* with the categorical product) then the unital axioms are $\lambda = \pi_1$ and $\rho = \pi_2$ (see definition of natural transformations for monoidal categories in terms of the categorical product in (B.1.3)), thus the following diagrams commute by the property that $\pi_1 \circ (f \times g) = f \circ \pi_1$ and $\pi_2 \circ (f \times g) = g \circ \pi_2$:

$$\begin{array}{ccc}
\mathsf{F}X \times 1 & \xleftarrow{id_{\mathsf{F}X} \times \mathsf{n}_1} & \mathsf{F}X \times \mathsf{F}1 \\
\downarrow{\scriptstyle \pi_1} & \swarrow^{\pi_1} & \mathsf{m}_{X,1} \uparrow\downarrow \mathsf{n}_{X,1} \\
\mathsf{F}X & \xleftarrow[\mathsf{F}\pi_1]{} & \mathsf{F}(X \times 1)
\end{array}
\qquad
\begin{array}{ccc}
1 \times \mathsf{F}X & \xleftarrow{\mathsf{n}_1 \times id_{\mathsf{F}X}} & \mathsf{F}1 \times \mathsf{F}X \\
\downarrow{\scriptstyle \pi_2} & \swarrow^{\pi_2} & \mathsf{m}_{X,1} \uparrow\downarrow \mathsf{n}_{1,X} \\
\mathsf{F}X & \xleftarrow[\mathsf{F}\pi_2]{} & \mathsf{F}(1 \times X)
\end{array}$$

Since F is strong, then n is reversed in the diagram with m. Thus, (B.1.6) follows.

□

B.1.3. Product/exponent equations

For the product/exponent adjunction $(- \times X) \dashv (X \Rightarrow -)$ where:

- $(- \times X)A = A \times X$ for all $A \in \mathcal{C}$
- $(- \times X)f = \; A \times X \xrightarrow{f \times id} B \times X$ for all $f : A \to B \in \mathcal{C}$
- $(X \Rightarrow -)A = X \Rightarrow A$ for all $A \in \mathcal{C}$
- $(X \Rightarrow -)f = \; X \Rightarrow A \xrightarrow{(f\circ)} X \Rightarrow B$ for all $f : A \to B \in \mathcal{C}$

Therefore, for all $g : B \to (X \Rightarrow X')$ and $f : A \to B$:

$$\begin{aligned} \phi^{-1}\,(g \circ f) &= \epsilon \circ (- \times X)(g \circ f) \\ &= \epsilon \circ ((- \times X)g) \circ ((- \times X)f) \\ &= (\phi^{-1}\; g) \circ (- \times X)f \\ &= (\phi^{-1}\; g) \circ (f \times id) \end{aligned} \qquad \begin{aligned} \phi\,(g \circ f) &= (X \Rightarrow -)(g \circ f) \circ \eta \\ &= (X \Rightarrow -)g \circ (X \Rightarrow -)f \circ \eta \\ &= (X \Rightarrow -)g \circ \phi\, f \end{aligned}$$

B.1.4. Strong functors and monads

Definition B.1.7. For a monoidal closed category $(\mathbb{C}, \otimes, I, \Rightarrow)$ an endofunctor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ is a *strong functor* if it has the natural transformation:

$$\mathsf{fmap}_{A,B} : (A \Rightarrow B) \to (\mathsf{F}A \Rightarrow \mathsf{F}B) \tag{76}$$

satisfying analogous coherence conditions to the normal functorial laws [**Koc72**][p.2]. Functorial strength corresponds to a kind of *internalised* morphism mapping for an endofunctor, where strong functors are the analogue of functors over *self-enriched categories*. [**Kel82**].

Kock showed that functorial strength corresponds to a notion of *tensorial strength* (*i.e.* they mutually define each other [**Koc72**][Theorem 1.3]):

Definition B.1.8. For a monoidal closed category, $(\mathbb{C}, \otimes, I, \Rightarrow)$ an endofunctor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ is a *tensorially strong functor* if it has a natural transformation:

$$\mathsf{st}_{A,B} : (A \otimes \mathsf{F}B) \to \mathsf{F}(A \otimes B) \tag{77}$$

and a number of coherence conditions [**Koc72**][p.3]:

$$\begin{array}{ccc} I \otimes \mathsf{M}A & \xrightarrow{\mathsf{st}_{I,A}} & \mathsf{M}(I \otimes A) \\ & \searrow^{\lambda_{\mathsf{M}A}} & \downarrow{\scriptstyle \mathsf{M}(\lambda_A)} \\ & & \mathsf{M}A \end{array} \qquad \begin{array}{ccccc} (A \otimes B) \otimes \mathsf{M}C & & \xrightarrow{\mathsf{st}_{A\otimes B,C}} & & \mathsf{M}((A \otimes B) \otimes C) \\ \downarrow{\scriptstyle \alpha_{A,B,\mathsf{M}C}} & & & & \downarrow{\scriptstyle \mathsf{M}(\alpha_{A,B,C})} \\ A \otimes (B \otimes \mathsf{M}C) & \xrightarrow[A \otimes \mathsf{st}_{B,C}]{} & A \otimes \mathsf{M}(B \otimes C) & \xrightarrow[\mathsf{st}_{A,B\otimes C}]{} & \mathsf{M}(A \otimes (B \otimes C)) \end{array} \tag{78}$$

For a functor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$ with tensorial strength, if $\mathbb{C}$ is a *symmetric* monoidal category, with $\gamma : A \otimes B \to B \otimes A$, then there is a symmetric notion, sometimes called *costrength*:

$$\mathsf{st}'_{A,B} = \; \mathsf{F}A \otimes B \xrightarrow{\gamma} B \otimes \mathsf{F}A \xrightarrow{\mathsf{st}_{A,B}} \mathsf{F}(B \otimes A) \xrightarrow{\mathsf{F}\gamma} \mathsf{F}(A \otimes B)$$

In this dissertation st' and st will be called *right strength* and *left strength* respectively.

Definition B.1.9. A *strong monad* comprises a monad (M, μ, η) and a *tensorial strength* $\mathsf{st} : (A \times \mathsf{M}B) \to \mathsf{M}(A \times B)$, satisfying a number of coherence conditions between the operations of the monad and st such that the following commute:

$$\begin{array}{ccccc} A \otimes \mathsf{MM}B & \xrightarrow{\mathsf{st}_{A,\mathsf{M}B}} & \mathsf{M}(A \otimes \mathsf{M}B) & \xrightarrow{\mathsf{M}(\mathsf{st}_{A,B})} & \mathsf{MM}(A \otimes B) \\ {\scriptstyle A\otimes\mu_B}\downarrow & & & & \downarrow{\scriptstyle \mu_{A\otimes B}} \\ A \otimes \mathsf{M}B & & \xrightarrow[\mathsf{st}_{A,B}]{} & & \mathsf{M}(A \otimes B) \end{array} \qquad \begin{array}{ccc} A \otimes B & \xrightarrow{id\otimes\eta_B} & A \otimes \mathsf{M}B \\ & {\scriptstyle \eta_{A\otimes B}}\searrow & \downarrow{\scriptstyle \mathsf{st}_{A,B}} \\ & & \mathsf{M}(A \otimes B) \end{array} \tag{79}$$

Lemma B.1.10. *A monad (M, μ, η) with endofunctor $\mathsf{M} : \mathbb{C} \to \mathbb{C}$ over a* CCC *is automatically a strong monad* [**Mog91**]*[p.19], since M is automatically a strong functor, with a canonical tensorial strength defined:*

$$\mathsf{sf} = \lambda(a, b).\mathsf{fmap}\ (\lambda b'.(a, b'))\ b$$

Proof of the coherence conditions for a strong monad is straightforward for this construction following from the laws of strong functors and the axioms of CCCs.

Kock also showed that given a strong monad M on a symmetric monoidal category (*i.e.* with left and right tensorial strengths) then there is a canonical definition of a *monoidal functor* structure for M [**Koc72**], with two possible definitions, one which "evaluates" (via strength) the first then second arguments, and the converse for the other.

B.1.5. Enriched categories, functors, and natural transformations

Lemma B.1.11. *Every closed monoidal category $(\mathbb{C}, \otimes, I, \Rightarrow, \phi : (- \otimes X \vdash X \Rightarrow -))$ is enriched over itself ($\mathbb{C}$-enriched) where:*

- *for all objects $A, B \in \mathbb{C}_0$, there is an object $A \Rightarrow B \in \mathbb{C}_0$ by* closedness*;*
- *for all objects $A, B, C \in \mathbb{C}_0$, the composition morphism M is defined:*

$$M_{A,B,C} = \phi\left(\mathsf{ev}_{B,C} \circ (id_{\mathbb{C}(B,C)} \otimes \mathsf{ev}_{A,B}) \circ \alpha_{\mathbb{C}(B,C),\mathbb{C}(A,B),A}\right) \tag{80}$$

- *for all objects $A \in \mathbb{C}_0$, the identity morphism I is: (where $\lambda_A : I \otimes A \to A$)*

$$I_A = \phi\left(\lambda_A\right) \tag{81}$$

The associativity and unital properties of M and I are easily proved by the properties of the adjunction and the laws of a symmetric, monoidal category.

Definition B.1.12. For $\mathbb{V}$-functors $\mathsf{F}, \mathsf{G} : \mathbb{C} \to \mathbb{D}$ a $\mathbb{V}$-*natural transformation* ($\mathbb{V}$-enriched natural transformation) α between F and G is a $\mathbb{C}_0$ indexed family of morphisms:

$$\alpha_A : I \to \mathbb{D}(\mathsf{F}\,A, \mathsf{G}\,A) \in \mathbb{V}_1$$

satisfying the $\mathbb{V}$-*naturality* condition:

$$M_{\mathsf{F}A,\mathsf{F}B,\mathsf{G}B}(\alpha_B \otimes \mathsf{F}_{A,B}f \circ \lambda^{-1}) = M_{\mathsf{F}A,\mathsf{G}A,\mathsf{G}B}(\mathsf{G}_{A,B}f \otimes \alpha_A \circ \rho^{-1}) \tag{82}$$

i.e.

$$\begin{array}{ccccc} & \xrightarrow{\lambda^{-1}} & I \otimes \mathbb{C}(A,B) & \xrightarrow{\alpha_B \otimes \mathsf{F}_{A,B}} & \mathbb{B}(\mathsf{F}B, \mathsf{G}B) \otimes \mathbb{B}(\mathsf{F}A, \mathsf{F}B) \\ \mathbb{C}(A,B) & & & & \xrightarrow{M_{\mathsf{F}A,\mathsf{F}B,\mathsf{G}B}} \mathbb{B}(\mathsf{F}A, \mathsf{G}B) \\ & \xrightarrow{\rho^{-1}} & \mathbb{C}(A,B) \otimes I & \xrightarrow[\mathsf{G}_{A,B} \otimes \alpha_A]{} & \mathbb{B}(\mathsf{G}A, \mathsf{G}B) \otimes \mathbb{B}(\mathsf{F}A, \mathsf{G}A) \xrightarrow{M_{\mathsf{F}A,\mathsf{G}A,\mathsf{G}B}} \end{array}$$

Definition B.1.13. For $\mathbb{V}$-functors $\mathsf{F}, \mathsf{G} : \mathbb{C} \to \mathbb{D}$ a $\mathbb{V}$-*natural transformation* ($\mathbb{V}$-enriched natural transformation) α between F and G is a $\mathbb{C}_0$ indexed family of morphisms:

$$\alpha_A : I \to \mathbb{D}(\mathsf{F}\,A, \mathsf{G}\,A) \in \mathbb{V}_1$$

satisfying a $\mathbb{V}$-*naturality* condition shown in Appendix B.1.5.

Lemma B.1.14. *In a self-enriched category $\mathbb{C}$ a $\mathbb{C}$-enriched natural transformation $\alpha_A : I \to \mathbb{C}(\mathsf{F}A, \mathsf{G}A) \in \mathbb{C}_1$ between $\mathbb{C}$-enriched endofunctors $\mathsf{F}, \mathsf{G} : \mathbb{C} \to \mathbb{C}$ is isomorphic to a natural transforation $\hat{\alpha} : \mathsf{F} \dot{\to} \mathsf{G}$, by the following isomorphism definition:*

$$\hat{\alpha}_A = \phi^{-1}(\alpha_A) \circ \lambda^{-1} : \mathsf{F}A \to \mathsf{G}A \in \mathbb{C}_1 \tag{83}$$

$$\alpha_A = \phi(\hat{\alpha}_A \circ \lambda) : I \to \mathbb{C}(\mathsf{F}A, \mathsf{G}A) \in \mathbb{C}_1 \tag{84}$$

where ϕ is the hom-set adjunction $\phi : \mathbb{C}(A \otimes B, C) \cong \mathbb{C}(A, B \Rightarrow C)$.

B.2. Comonads

B.2.1. Tree zipper calculation

Using McBride's differentation technique [**McB01**, **AAMG04**], the binary tree zipper for $\mathsf{Tree}\,A = \mu X.A + A \times X^2$ can be calculated as follows:

$$\begin{aligned} \partial_A(\mathsf{Tree}\,A) = &\ \mu Z.\partial_A(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\,A]} + \partial_X(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\,A]} \times Z \\ = &\ \mu Z.1 + (\partial_A A \times X^2 + A \times \partial_A X^2)_{[X \mapsto \mathsf{Tree}\,A]} + \partial_X(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\,A]} \times Z \\ = &\ \mu Z.1 + (1 \times X^2 + A \times 0)_{[X \mapsto \mathsf{Tree}\,A]} + \partial_X(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\,A]} \times Z \\ = &\ \mu Z.1 + (\mathsf{Tree}\,A)^2 + \partial_X(A + A \times X^2)_{[X \mapsto \mathsf{Tree}\,A]} \times Z \\ = &\ \mu Z.1 + (\mathsf{Tree}\,A)^2 + (0 + \partial_X A \times X^2 + A \times \partial_X X^2)_{[X \mapsto \mathsf{Tree}\,A]} \times Z \\ = &\ \mu Z.1 + (\mathsf{Tree}\,A)^2 + (0 + 0 + A \times 2 \times X)_{[X \mapsto \mathsf{Tree}\,A]} \times Z \\ = &\ \mu Z.1 + (\mathsf{Tree}\,A)^2 + A \times 2 \times \mathsf{Tree}\,A \times Z \\ = &\ (\mu Z.1 + A \times 2 \times \mathsf{Tree}\,A \times Z) \times (1 + (\mathsf{Tree}\,A)^2) \\ = &\ (\mathsf{Path}\,A) \times (1 + (\mathsf{Tree}\,A)^2) \end{aligned}$$

Therefore a comonad for this zipper is given by:

$$\begin{aligned} A \times \partial_A(\mathsf{Tree}\,A) = &\ A \times (\mathsf{Path}\,A) \times (1 + (\mathsf{Tree}\,A)^2) \\ = &\ (\mathsf{Path}\,A) \times (A + A \times (\mathsf{Tree}\,A)^2) \\ = &\ (\mathsf{Path}\,A) \times (\mathsf{Tree}\,A) \end{aligned}$$

B.2.2. Non-empty list zipper calculation

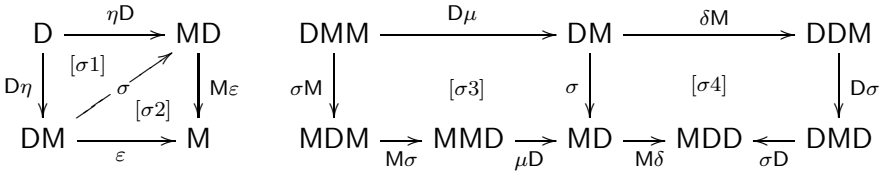
The non-empty list zipper for $\mathsf{NEList}\,A = \mu X.A + A \times X$ is calculated as follows, where $\mathsf{List}\,A = \mu X.1 + A \times X$:

$$\begin{aligned}
\partial_A(\mathsf{NEList}\,A) &\cong \partial_A(\mu X.A + A \times X) \\
&\cong \mu Z.\partial_A(A + A \times X)_{[X \mapsto \mathsf{NEList}\,A]} + \partial_X(A + A \times X)_{[X \mapsto \mathsf{NEList}\,A]} \times Z \\
&\cong \mu Z.(1 + (\partial_A A \times X + A \times \partial_A X))_{[X \mapsto \mathsf{NEList}\,A]} + \partial_X(A + A \times X)_{[X \mapsto \mathsf{NEList}\,A]} \times Z \\
&\cong \mu Z.(1 + X)_{[X \mapsto \mathsf{NEList}\,A]} + \partial_X(A + A \times X)_{[X \mapsto \mathsf{NEList}\,A]} \times Z \\
&\cong \mu Z.1 + (\mathsf{NEList}\,A) + (0 + (\partial_X A \times X + A \times \partial_X X))_{[X \mapsto \mathsf{NEList}\,A]} \times Z \\
&\cong \mu Z.1 + (\mathsf{NEList}\,A) + A \times Z \\
&\cong (\mu Z.1 + A \times Z) \times (1 + (\mathsf{NEList}\,A)) \\
&\cong (\mathsf{List}\,A) \times (\mathsf{List}\,A)
\end{aligned}$$

Therefore a comonad for this zipper is: $\mathsf{NEList}'A = A \times \partial_A(\mathsf{NEList}A)$ which is the pair of a non-empty list with a possibly-empty list (mentioned by Ahman *et al.* [**ACU12**]).

B.3. Generalising comonads

Definition B.3.1. For a monad (M, η, μ) and a comonad $(\mathsf{D}, \varepsilon, \delta)$ a distributive law of D over M is the natural transformation $\sigma : \mathsf{DM} \dot{\to} \mathsf{MD}$ with the following axioms on the interaction of σ with η, ε, μ, and δ [**BS93**]:



B.4. The coeffect calculus

B.4.1. Coeffects for equational theories of let-binding

Associativity.

$$[\textsc{let}]\ \frac{\Gamma \,?\, X \vdash e_1 : \tau_1 \qquad [\textsc{let}]\ \dfrac{\Gamma, x : \tau_1 \,?\, Y \vdash e_2 : \tau_2 \qquad [\textsc{exchg}]\ \dfrac{[\textsc{weak}]\ \dfrac{\Gamma, y : \tau_2 \,?\, Z \vdash e_3 : \tau_3}{\Gamma, y : \tau_2, x : \tau_1 \,?\, Z \vdash e_3 : \tau_3}}{\Gamma, x : \tau_1, y : \tau_2 \,?\, Z \vdash e_3 : \tau_3}}{\Gamma, x : \tau_1 \,?\, (Z \sqcup (Z \bullet Y)) \vdash \mathbf{let}\ y = e_2\ \mathbf{in}\ e_3 : \tau_3}}{\Gamma \,?\, ((Z \sqcup (Z \bullet Y)) \sqcup ((Z \sqcup (Z \bullet Y)) \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ (\mathbf{let}\ y = e_2\ \mathbf{in}\ e_3) : \tau_3}$$

$$\equiv \quad [\textsc{let}]\ \frac{[\textsc{let}]\ \dfrac{\Gamma \,?\, X \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \,?\, Y \vdash e_2 : \tau_2}{\Gamma \,?\, (Y \sqcup (Y \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2} \qquad \Gamma, y : \tau_2 \,?\, Z \vdash e_3 : \tau_3}{\Gamma \,?\, (Z \sqcup (Z \bullet (Y \sqcup (Y \bullet X)))) \vdash \mathbf{let}\ y = (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2)\ \mathbf{in}\ e_3 : \tau_3}$$

Left unit.

$$[\textsc{let}]\ \frac{\Gamma\,?\,X \vdash e_1 : \tau_1 \qquad [\textsc{var}]\ \dfrac{x : \tau_1 \in \Gamma, x : \tau_1}{\Gamma, x : \tau_1\,?\,I \vdash x : \tau_1}}{\Gamma\,?\,(I \sqcup (I \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ x : \tau_1} \quad \equiv \quad \Gamma\,?\,X \vdash e_1 : \tau_1$$

Right unit.

$$\begin{array}{c}[\textsc{var}]\ \dfrac{y : \tau_1 \in (\Gamma, y : \tau_1)}{\Gamma, y : \tau_1\,?\,I \vdash y : \tau_1} \qquad [\textsc{weak}]\ \dfrac{\Gamma, y : \tau_1\,?\,Y \vdash e_2 : \tau_2}{\Gamma, y : \tau_1, x : \tau_1\,?\,Y \vdash e_2 : \tau_2} \\ [\textsc{let}]\ \dfrac{}{\Gamma, y : \tau_1\,?\,(Y \sqcup (Y \bullet I)) \vdash \mathbf{let}\ x = y\ \mathbf{in}\ e_2 : \tau_2}\end{array}$$

$$\equiv \qquad [\alpha]\ \frac{\Gamma, x : \tau_1\,?\,Y \vdash e_2 : \tau_2}{\Gamma, y : \tau_1\,?\,Y \vdash e_2[x := y] : \tau_2}$$

Weakening.

$$[\textsc{let}]\ \frac{\Gamma\,?\,X \vdash e_1 : \tau_1 \qquad [\textsc{weaken}]\ \dfrac{\Gamma\,?\,Y \vdash e_2 : \tau_2}{\Gamma, x : \tau_1\,?\,Y \vdash e_2 : \tau_2}}{\Gamma\,?\,(Y \sqcup (Y \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2} \quad \equiv \quad \Gamma\,?\,Y \vdash e_2 : \tau_2 \quad (x \notin FV(e_2))$$

Contraction.

$$[\textsc{let}]\ \frac{\Gamma\,?\,X \vdash e_1 : \tau_1 \qquad \begin{array}{c}[\textsc{weaken}]\ \dfrac{\Gamma\,?\,X \vdash e_1 : \tau_1}{\Gamma, x : \tau_1\,?\,X \vdash e_1 : \tau_1} \qquad \Gamma, x : \tau_1, y : \tau_1\,?\,Z \vdash e_2 : \tau_2 \\ [\textsc{let}]\ \dfrac{}{\Gamma, x : \tau_1\,?\,(Z \sqcup (Z \bullet X)) \vdash \mathbf{let}\ y = e_1\ \mathbf{in}\ e_2 : \tau_2}\end{array}}{\Gamma\,?\,((Z \sqcup (Z \bullet X)) \sqcup ((Z \sqcup (Z \bullet X)) \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ (\mathbf{let}\ y = e_1\ \mathbf{in}\ e_2) : \tau_2}$$

$$\equiv \qquad [\textsc{let}]\ \frac{\Gamma\,?\,X \vdash e_1 : \tau_1 \qquad [\textsc{contr}]\ \dfrac{\Gamma, x : \tau_1, y : \tau_1\,?\,Z \vdash e_2 : \tau_2}{\Gamma, z : \tau_1\,?\,Z \vdash e_2[y := z][x := z] : \tau_2}}{\Gamma\,?\,(Z \sqcup (Z \bullet X)) \vdash \mathbf{let}\ z = e_1\ \mathbf{in}\ e_2[z := y][z := x] : \tau_2}$$

Exchange.

$$[\textsc{let}]\ \frac{\Gamma\,?\,X \vdash e_1 : \tau_1 \qquad \begin{array}{c}[\textsc{weaken}]\ \dfrac{\Gamma\,?\,Y \vdash e_2 : \tau_2}{\Gamma, x : \tau_1\,?\,Y \vdash e_2 : \tau_2} \qquad \Gamma, x : \tau_1, y : \tau_2\,?\,Z \vdash e_3 : \tau_3 \\ [\textsc{let}]\ \dfrac{}{\Gamma, x : \tau_1\,?\,(Z \sqcup (Z \bullet Y)) \vdash \mathbf{let}\ y = e_2\ \mathbf{in}\ e_3 : \tau_3}\end{array}}{\Gamma\,?\,((Z \sqcup (Z \bullet Y)) \sqcup ((Z \sqcup (Z \bullet Y)) \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ (\mathbf{let}\ y = e_2\ \mathbf{in}\ e_3) : \tau_3}$$

$$\equiv \qquad [\textsc{let}]\ \frac{\Gamma\,?\,Y \vdash e_2 : \tau_2 \qquad \begin{array}{c}[\textsc{weaken}]\ \dfrac{\Gamma\,?\,X \vdash e_1 : \tau_1}{\Gamma, y : \tau_2\,?\,X \vdash e_1 : \tau_1} \qquad [\textsc{exchg}]\ \dfrac{\Gamma, x : \tau_1, y : \tau_2\,?\,Z \vdash e_3 : \tau_3}{\Gamma, y : \tau_2, x : \tau_1\,?\,Z \vdash e_3 : \tau_3} \\ [\textsc{let}]\ \dfrac{}{\Gamma, y : \tau_2\,?\,(Z \sqcup (Z \bullet X)) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_3 : \tau_3}\end{array}}{\Gamma\,?\,((Z \sqcup (Z \bullet X)) \sqcup ((Z \sqcup (Z \bullet X)) \bullet Y)) \vdash \mathbf{let}\ y = e_2\ \mathbf{in}\ (\mathbf{let}\ x = e_1\ \mathbf{in}\ e_3) : \tau_3}$$

B.4.2. Traditional semantics for the coeffect calculus

Section 6.4.1 developed a semantics for the coeffect calculus using the indexed comonad structure and following the categorical semantisc appraoch for the simply-typed λ-calculus, shown in Chapter 2. The following shows diagrams for the semantics of [APP] and [LET] to aid understanding of the coeffect's derviation.

– [APP] for $k_1 : \mathsf{D}_R\Gamma \to (\mathsf{D}_S\sigma \Rightarrow \tau)$ and $k_2 : \mathsf{D}_T\Gamma \to \sigma$, the semantics is as follows (split across two lines due to space constraints).

$$\mathsf{D}_{I\sqcup T}\Gamma \xrightarrow{\mathsf{n}_{I,T}\circ\mathsf{D}\Delta} \mathsf{D}_I\Gamma \times \mathsf{D}_T\Gamma \xrightarrow{\varepsilon\times k_2} \Gamma \times \sigma \quad (85)$$

$$\mathsf{D}_{(R\sqcup S)\bullet(I\sqcup T)}\Gamma \xrightarrow{\delta_{(R\sqcup S),(I\sqcup T)}} \mathsf{D}_{R\sqcup S}\mathsf{D}_{I\sqcup T}\Gamma \xrightarrow{\mathsf{D}(eq.(85))} \mathsf{D}_{R\sqcup S}(\Gamma \times \sigma) \xrightarrow{\mathsf{n}_{R,S}} \mathsf{D}_R\Gamma \times \mathsf{D}_S\sigma \xrightarrow{\psi k_1} \tau \quad (86)$$

– [LET] for $k_2 : \mathsf{D}_S(\Gamma \times \sigma) \to \tau$ and $k_1 : \mathsf{D}_R\Gamma \to \sigma$ then:

$$\mathsf{D}_{S\bullet(I\sqcup R)}\Gamma \xrightarrow{\delta_{S,(I\sqcup R)}} \mathsf{D}_S\mathsf{D}_{I\sqcup R}\Gamma \xrightarrow{\mathsf{D}(\mathsf{n}_{I,R}\circ\mathsf{D}\Delta)} \mathsf{D}_S(\mathsf{D}_I\Gamma \times \mathsf{D}_R\Gamma) \xrightarrow{\mathsf{D}(\varepsilon\times k_1)} \mathsf{D}_S(\Gamma \times \sigma) \xrightarrow{k_2} \tau \quad (87)$$

B.5. A contextual language for containers

B.5.1. Final coalgebras

Definition B.5.1. A *coalgebra homomorphism* between two F-coalgebras $(U, c : U \to \mathsf{F}U)$ and $(V, d : V \to \mathsf{F}V)$, for the endofunctor $\mathsf{F} : \mathbb{C} \to \mathbb{C}$, is a morphism $f : U \to V \in \mathbb{C}_1$ such that the following diagram commutes:

$$\begin{array}{ccc} U & \xrightarrow{f} & V \\ \downarrow{\scriptstyle c} & & \downarrow{\scriptstyle d} \\ \mathsf{F}U & \xrightarrow[\mathsf{F}f]{} & \mathsf{F}V \end{array} \quad (88)$$

Definition B.5.2. An F-coalgebra $(V, d : V \to \mathsf{F}V)$ is *final* if for all F-coalgebras $(U, c : U \to \mathsf{F}U)$ there is a unique coalgebra homomorphism $f : U \to V$.

Lemma B.5.3. *The final coalgebra $(V, d : V \to \mathsf{F}V)$ has as its object the greatest fixed point of the functor $\mathsf{F}V \cong V$ of the functor F, and if it exists, is unique up-to-isomorphism.* [**JR97**].

Existence of the final coalgebra gives a principle of definition by *coinduction*, and its uniqueness gives a principle of proof by coinduction. For definition of functions by coinduction we define the coalgebra operations for the arbitrary F-coalgebra on U, then the unique homomorphism proceeds by coinduction, capturing the observations that arise by repeatedly applying the "next step" operation. unique homomorphism is defined.

Example B.5.4. The final coalgebra for the endofunctor $\mathsf{F}\,X = A \times X$has as its carrier object the type of streams on A, $A^{\mathbb{N}}$ with coalgebra morphism $\langle \mathsf{value}, \mathsf{next}\rangle s = (s0, \lambda n.\, s(n+1))$.

The final coalgebra for the functor $FU = A \times U \times U$ captures infinite binary trees with carrier object A^{2^*} where 2^* are finite sequences of tags pointing to the left or right subtree. Deconstructors on a list-zipper, of element type A, are coalgebras $\langle \mathsf{current}, \mathsf{left}, \mathsf{right}\rangle : \mathit{ListZipper}\,A \to A \times (\mathit{ListZipper}\,A) \times (\mathit{ListZipper}\,A)$ *i.e.* for $FU = A \times U \times U$, where the first component of the product is the current element, the second component is the list to the left of the current element and the third component is the list to the right of the current element. Thus, the mapping of the list-zipper coalgebra to the final coalgebra computes the corresponding infinite binary trees.

Appendix C

Additional Proofs

C.1. Background

C.1.1. η- and extensional-equality for functions

For the simply-typed λ-calculus, the [$\equiv$-η] (η equality) and extensional equality [EXT]:

$$[\equiv\text{-}\eta]\frac{\Gamma \vdash e : \sigma \Rightarrow \tau}{\Gamma \vdash (\lambda x.e\ x) \equiv e : \sigma \Rightarrow \tau}\ x \notin FV e \qquad [\text{ext}]\frac{\Gamma, x : \tau \vdash e\ x \equiv e'\ x : \tau'}{\Gamma \vdash e \equiv e' : \tau \Rightarrow \tau'}\ x \notin FV(e), x \notin FV(e')$$

are equivalent, *i.e.* [EXT] is admissible in the presence of [$\equiv$-η], and vice versa, as shown by the following mutual derivations which shows η-equivalence implies extensionality of functions:

$$\Delta = [\text{trans}]\frac{[\lambda\text{-cong}]\ \dfrac{\Gamma, x : \tau \vdash e\ x \equiv e'\ x : \tau'}{\Gamma \vdash \lambda x.e\ x \equiv \lambda x.e'\ x : \tau \Rightarrow \tau'} \qquad [\eta]\ \dfrac{}{\Gamma \vdash \lambda x.e\ x \equiv e : \tau \Rightarrow \tau'}}{\Gamma \vdash e \equiv \lambda x.e' x : \tau \Rightarrow \tau'}$$

$$[\text{trans}]\frac{[\eta]\ \dfrac{}{\Gamma \vdash \lambda x.e\ x \equiv e' : \tau \Rightarrow \tau'} \qquad \Delta}{\Gamma \vdash e \equiv e' : \tau \Rightarrow \tau'} \cong [\text{ext}] \tag{89}$$

$$[\text{ext}]\frac{[\beta]\ \dfrac{\Gamma \vdash e : \tau \Rightarrow \tau'}{\Gamma, x : \tau \vdash (\lambda x.e\ x)\ x \equiv e\ x : \tau'}}{\Gamma \vdash \lambda x.e\ x \equiv e : \tau \Rightarrow \tau'} \cong [\equiv\text{-}\eta] \tag{90}$$

C.1.2. Proof of [$\equiv$-β] and [$\equiv$-η] for the simply-typed λ-calculus

The following gives the proofs of β- and η-equality for the categorical semantics of the simply-typed λ-calculus (in terms of a CCC) for Chapter 2.

$$\begin{aligned}
&[\![\Gamma \vdash \lambda x.e\ x : \tau \Rightarrow \tau']\!] \\
=\ &\lambda\left(\mathsf{ev} \circ \langle [\![\Gamma, x : \tau \vdash e : \tau \Rightarrow \tau']\!], [\![\Gamma, x : \tau \vdash x : \tau]\!]\rangle\right) && [\![\text{ABS}]\!], [\![\text{APP}]\!] \\
=\ &\lambda\left(\mathsf{ev} \circ \langle [\![\Gamma \vdash e : \tau \Rightarrow \tau']\!] \circ \pi_1, [\![\Gamma, x : \tau \vdash x : \tau]\!]\rangle\right) && [\![\text{WEAKEN}]\!] \\
=\ &\lambda\left(\mathsf{ev} \circ \langle [\![\Gamma \vdash e : \tau \Rightarrow \tau']\!] \circ \pi_1, \pi_2\rangle\right) && [\![\text{VAR}]\!] \\
=\ &\lambda\left(\mathsf{ev} \circ ([\![\Gamma \vdash e : \tau \Rightarrow \tau']\!] \times id) \circ \langle \pi_1, \pi_2\rangle\right) && \{by \times \textit{universal property}\} \\
=\ &\lambda\left(\mathsf{ev} \circ ([\![\Gamma \vdash e : \tau \Rightarrow \tau']\!] \times id)\right) && \{\times \textit{universal property}\} \\
=\ &\lambda\left(\mathsf{ev} \circ (\lambda f' \times id)\right) && \{[\![\Gamma \vdash e : \tau \Rightarrow \tau']\!] = \lambda f'\} \\
=\ &\lambda\, f' && \{\Rightarrow \textit{universal property}\} \\
=\ &[\![\Gamma \vdash e : \tau \Rightarrow \tau']\!] && \{[\![\Gamma \vdash e : \tau \Rightarrow \tau']\!] = \lambda f'\}\ \square
\end{aligned}$$

$$
\begin{aligned}
& [\![\Gamma \vdash (\lambda x.\, e_1)\, e_2 : \tau']\!] \\
= \;& \mathsf{ev} \circ \langle \lambda[\![\Gamma, x : \tau \vdash e_1 : \tau']\!], [\![\Gamma \vdash e_2 : \tau]\!] \rangle && [\![\textsc{app}]\!], [\![\textsc{abs}]\!] \\
= \;& \mathsf{ev} \circ (\lambda[\![\Gamma, x : \tau \vdash e_1 : \tau']\!] \times id) \circ \langle id, [\![\Gamma \vdash e_2 : \tau]\!] \rangle && \{\times \textit{ universal property}\} \\
= \;& [\![\Gamma, x : \tau \vdash e_1 : \tau']\!] \circ \langle id, [\![\Gamma \vdash e_2 : \tau]\!] \rangle && \{\Rightarrow \textit{ universal property}\} \\
= \;& [\![\Gamma \vdash \mathbf{let}\, x = e_2\, \mathbf{in}\, e_1 : \tau']\!] && [\![\mathbf{let}]\!] \\
= \;& [\![\Gamma \vdash e_1[x := e_2] : \tau']\!] && \{\textit{let-}\beta \textit{ - Lemma 2.2.5}\}\ \square
\end{aligned}
$$

C.1.3. Proof that the product/exponent adjunction implies categorical exponents

Lemma C.1.1. *Given* $(- \times X) \dashv (X \Rightarrow -)$, $\Rightarrow$ *is a categorical exponent where* $\mathsf{ev} = \epsilon$, $\lambda f = \phi f$.

Proof. From this construction, tthe universal property of $\Rightarrow$ holds with the following proof (where $\mathsf{L} = (- \times X)$ and $\mathsf{R} = (X \Rightarrow -)$ in $\mathsf{L} \dashv \mathsf{R}$):

$$
\begin{aligned}
& \mathsf{ev} \circ (\lambda g \times id) \\
= \;& \epsilon \circ (\phi g \times id) && \{\textit{Lemma 2.2.17}\} \\
= \;& \epsilon \circ \mathsf{L}(\phi g) && \{\textit{definition of } \mathsf{L}\} \\
= \;& \epsilon \circ \mathsf{LR}g \circ \mathsf{L}\eta && (2.2.16) \\
= \;& g \circ \epsilon\mathsf{L} \circ \mathsf{L}\eta && \{\textit{naturality of } \epsilon\} \\
= \;& g && [A1] \quad \square
\end{aligned}
$$

Furthermore the property that $\phi g : X \to (Y \Rightarrow Z)$, for $g : X \times Y \to Z$, is the unique morphism which satisfies the universal property is proved by

$\square$

C.2. Comonads

C.2.1. Properties of lax and colax monoidal functors and comonads

Lemma 3.3.5. (originally on p. 66) Every endofunctor F on a category with finite products has a canonical *unique* colax monoidal structure with $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$ and $\mathsf{n}_1 = !_{\mathsf{F}1}$.

Proof. It is straightforward to prove the coherence conditions of a colax monoidal functor for $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$ and $\mathsf{n}_1 = !_{\mathsf{F}1}$ (see Definition B.1.5 (p. 187), for these conditions) which follow from universal properties of products and the definitions of the natural transformations for the monoidal category. Since the underlying category $\mathbb{C}$ has categorical products, these natural transformations are the canonical definitions shown in Example B.1.3 (p.186).

To prove uniqueness of these definitions, assume the existence of some other colax monoidal structure $\mathsf{n}'_{A,B}$ and n'_1. Since the underlying category $\mathbb{C}$ has categorical products, the natural transformations providing units for products are uniquely $\lambda : 1 \times A \to A = \pi_2$ and $\rho : A \times 1 \to$

$A = \pi_1$ (Example B.1.3 [p.186]). The unital properties of n'_1 with respect to $\mathsf{n}'_{A,B}$, are therefore:

$$\begin{array}{ccccc} \mathsf{F}X \times 1 & \xleftarrow{id\times \mathsf{n}'_1} & \mathsf{F}X \times \mathsf{F}1 & \quad \mathsf{F}1 \times \mathsf{F}X & \xrightarrow{\mathsf{n}'_1 \times id} & 1 \times \mathsf{F}X \\ \downarrow{\scriptstyle \pi_1} & \swarrow{\scriptstyle \pi_1} & \uparrow{\scriptstyle \mathsf{n}'_{X,1}} & \quad \uparrow{\scriptstyle \mathsf{n}'_{1,X}} & \searrow{\scriptstyle \pi_2} & \downarrow{\scriptstyle \pi_2} \\ \mathsf{F}X & \xleftarrow[\mathsf{F}\pi_1]{} & \mathsf{F}(X \times 1) & \quad \mathsf{F}(1 \times X) & \xrightarrow[\mathsf{F}\pi_2]{} & \mathsf{F}X \end{array} \tag{91}$$

where the diagonal arrows follow from the universal property of products.

From the universal property of products, $f_1 = g_1, f_2 = g_2 \Rightarrow \langle f_1, f_2\rangle = \langle g_1, g_2\rangle$, therefore:

$$\begin{array}{lcll} \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2\rangle & = & \langle \pi_1 \circ \mathsf{n}'_{X,Y}, \pi_2 \circ \mathsf{n}'_{X,Y}\rangle & \{\textit{pairing of diagrams from (91)}\} \\ & = & \langle \pi_1, \pi_2\rangle \circ \mathsf{n}'_{X,Y} & \{\times \textit{ universal property }\} \\ & = & \mathsf{n}'_{X,Y} & \{\times \textit{ universal property}\} \end{array}$$

Therefore, by contradiction of the distinctness of $\mathsf{n}'_{X,Y}$ and $\mathsf{n}_{X,Y}$, the canonical colax definition $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2\rangle$ is unique. The operation $\mathsf{n}_1 = !_{\mathsf{F}1}$ is trivially unique by the universal property of the terminal morphism.

□

Proposition C.2.1. *For an endofunctor* F *over a category with finite products, the following two properties are equivalent:*

(i) $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2\rangle$
(ii) $\mathsf{n}_{A,B} \circ \mathsf{F}\Delta = \Delta$ *(i.e.* F *is idempotent colax semi-monoidal)*

Proof. Idempotence (ii) follows from (i) as follows:

$$\begin{array}{lcll} \mathsf{n}_{A,B} \circ \mathsf{F}\Delta & = & \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2\rangle \circ \mathsf{F}\Delta & \text{(ii)} \\ & = & \langle \mathsf{F}(\pi_1 \circ \Delta), \mathsf{F}(\pi_2 \circ \Delta)\rangle & \{\langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle \textit{ and } \mathsf{F} \textit{ functoriality}\} \\ & = & \langle \mathsf{F}id, \mathsf{F}id\rangle & \{\times \textit{ universal property}\} \\ & = & \Delta & \{\Delta = \langle id, id\rangle\} \end{array}$$

The canonical $\mathsf{n}_{A,B}$ definition (i) follows from idempotence (ii):

$$\begin{array}{lcll} \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2\rangle = & (\mathsf{F}\pi_1 \times \mathsf{F}\pi_2) \circ \Delta & \{\textit{universal property of } \times\} \\ & = & (\mathsf{F}\pi_1 \times \mathsf{F}\pi_2) \circ \mathsf{n}_{A,B} \circ \mathsf{F}\Delta & \text{(i)} \\ & = & \mathsf{n}_{A,B} \circ \mathsf{F}(\pi_1 \times \pi_2) \circ \mathsf{F}\Delta & \{\mathsf{n}_{A,B} \textit{ naturality}\} \\ & = & \mathsf{n}_{A,B} & \{\mathsf{F} \textit{ functoriality and } \times \textit{ universal property}\} \end{array}$$

□

Proposition C.2.2. *For a lax (semi-)monoidal endofunctor* F *on a category with finite products, and canonical colax operation* $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2\rangle$ *the following two properties are equivalent:*

(i) $\mathsf{F}\Delta = \mathsf{m}_{A,B} \circ \Delta$ *(i.e.* F *is idempotent lax monoidal);*
(ii) $\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id$.

Proof. (i) ⇒ (ii) is as follows:

$$\begin{aligned}
\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} &= \mathsf{m}_{A,B} \circ \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle && \{\textit{canonical } \mathsf{n}_{A,B} \textit{ definition}\} \\
&= \mathsf{m}_{A,B} \circ (\mathsf{F}\pi_1 \times \mathsf{F}\pi_2) \circ \Delta && \{\times \textit{ universal property }\} \\
&= \mathsf{F}(\pi_1 \times \pi_2) \circ \mathsf{m}_{A,B} \circ \Delta && \{\mathsf{m}_{A,B} \textit{ naturality}\} \\
&= \mathsf{F}(\pi_1 \times \pi_2) \circ \mathsf{F}\Delta && \{\mathsf{m}_{A,B} \textit{ idempotence (31)}\} \\
&= id && \{\mathsf{F} \textit{ functoriality, } \times \textit{ universal property}\}
\end{aligned}$$

(ii) ⇒ (i) follows by the following proof:

$$\begin{aligned}
\mathsf{F}\Delta &= \mathsf{F}\Delta && \{\textit{taut.}\} \\
&= \mathsf{m}_{A,B} \circ \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle \circ \mathsf{F}\Delta && \{\mathsf{m}_{A,B} \circ \mathsf{n}_{A,B} = id \textit{ and canonical } \mathsf{n}_{A,B}\} \\
&= \mathsf{m}_{A,B} \circ \langle \mathsf{F}id, \mathsf{F}id \rangle && \{\mathsf{F} \textit{ functoriality, } \times \textit{ universal property}\} \\
&= \mathsf{m}_{A,B} \circ \Delta && \{\Delta = \langle id, id \rangle\}
\end{aligned}$$

□

Proposition C.2.3. *For a lax (semi-)monoidal endofunctor* F *on a category with finite products and canonical colax operation* $\mathsf{n}_{A,B} = \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle$*, the following two properties are equivalent:*

(i) $\mathsf{F}\pi_1 \circ \mathsf{m}_{A,B} = \pi_1$ *and* $\mathsf{F}\pi_2 \circ \mathsf{m}_{A,B} = \pi_2$*;*
(ii) $\mathsf{n}_{A,B} \circ \mathsf{m}_{A,B} = id$*.*

Proof. (ii) ⇒ (i) (where the proof is symmetric for π_1 and π_2, only that for π_1 is shown):

$$\begin{aligned}
id &= \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle \circ \mathsf{m}_{A,B} && \{\mathsf{n}_{A,B} \textit{ is left-inverse to } \mathsf{m}_{A,B}\} \\
\pi_1 &= \pi_1 \circ \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle \circ \mathsf{m}_{A,B} && \{\textit{post-compose } \pi_1\} \\
&= \mathsf{F}\pi_1 \circ \mathsf{m}_{A,B} && \{\times \textit{ universal property}\}
\end{aligned}$$

$$\begin{aligned}
\text{(i) ⇒ (ii)}: \quad \langle \pi_1, \pi_2 \rangle &= \langle \mathsf{F}\pi_1 \circ \mathsf{m}_{A,B}, \mathsf{F}\pi_2 \circ \mathsf{m}_{A,B} \rangle && \{\textit{from (i) and by } \times \textit{ universal property}\} \\
id &= \langle \mathsf{F}\pi_1, \mathsf{F}\pi_2 \rangle \circ \mathsf{m}_{A,B} && \{\times \textit{ universal property}\}
\end{aligned}$$

□

C.2.2. η-equality for comonadic semantics

For η-equality, one of the following must hold (Section 2.2.7, *((B.iii))*):

$$\Phi\,(g \circ^{\mathsf{D}} f) = (\Rightarrow^{\mathsf{D}} g) \circ^{\mathsf{D}} \Phi f \qquad (\textit{where } f : \mathsf{D}(A \times B) \to X, g : \mathsf{D}X \to Y) \tag{92}$$

$$\Psi(g \circ^{\mathsf{D}} f) = (\Psi g) \circ^{\mathsf{D}} (f \times^{\mathsf{D}} id^{\mathsf{D}}) \qquad (\textit{where } f : \mathsf{D}A \to X, g : \mathsf{D}X \to (\mathsf{D}B \Rightarrow Y)) \tag{93}$$

Subsequently, either $\times^{\mathsf{D}}$ or $\Rightarrow^{\mathsf{D}}$ must be defined as a bifunctor (not just an object mapping). For (93), the bifunctor $(- \times^{\mathsf{D}} -) : \mathbb{C}_{\mathsf{D}} \times \mathbb{C}_{\mathsf{D}} \to \mathbb{C}_{\mathsf{D}}$ is defined for objects and morphisms as:

- $(A \times^{D} B) = A \times B \in \mathbb{C}_{\mathsf{D}0}$
- $(f \times^{D} g) = (f \times g) \circ \mathsf{n}_{A,B} : A \times^{D} B \to X \times^{D} Y \in \mathbb{C}_{\mathsf{D}1}$ (*i.e.*$f : \mathsf{D}A \to X, g : \mathsf{D}B \to Y \in \mathbb{C}_1$ and $(f \times^{D} g) : \mathsf{D}(A \times B) \to X \times Y \in \mathbb{C}_1$)

Proof of (93) is then as follows, requiring that δ is a colax monoidal natural transformation:

$$\begin{aligned}
(\Psi g) \circ^{\mathsf{D}} (f \times^{\mathsf{D}} id^{\mathsf{D}}) &= \psi g \circ \mathsf{n}_{Y,A} \circ ((f \times \varepsilon) \circ \mathsf{n}_{X,A})^\dagger && \{\Psi \textit{ definition}/id^{\mathsf{D}}/\times^{\mathsf{D}}/\circ^{\mathsf{D}}\} \\
&= \psi g \circ \mathsf{n}_{Y,A} \circ \mathsf{D}(f \times \varepsilon) \circ \mathsf{D}\mathsf{n}_{X,A} \circ \delta && \{\mathsf{D}f \circ \delta = f^\dagger\} \\
&= \psi g \circ (\mathsf{D}f \times \mathsf{D}\varepsilon) \circ \mathsf{n}_{Y,A} \circ \mathsf{D}\mathsf{n}_{X,A} \circ \delta && \{\mathsf{n}_{A,B} \textit{ naturality}\} \\
&= \psi g \circ (\mathsf{D}f \times \mathsf{D}\varepsilon) \circ (\delta \times \delta) \circ \mathsf{n}_{X,A} && \{\delta \textit{ colax monoidal } (33)\} \\
&= \psi g \circ (\mathsf{D}f \circ \delta \times \mathsf{D}\varepsilon \circ \delta) \circ \mathsf{n}_{X,A} && \{- \times - \textit{ functoriality}\} \\
&= \psi g \circ (f^\dagger \times id) \circ \mathsf{n}_{X,A} && \{\mathsf{D}f \circ \delta = f^\dagger \textit{ and } [C2]\} \\
&= \psi(g \circ f^\dagger) \circ \mathsf{n}_{X,A} && \{\psi(g \circ f) = \psi g \circ (f \times id)\} \\
&= \Psi(g \circ^{\mathsf{D}} f) && \{\Psi \textit{ definition}/\circ^{\mathsf{D}}\} \quad \square
\end{aligned}$$

C.2.3. β-equivalence for comonadic semantics

Lemma C.2.4. *((let-β) - Substitution equivalent to* **let***)*

$$\forall e, e'.\ (\Gamma, x : \tau' \vdash e : \tau) \wedge (\Gamma \vdash e' : \tau') \Rightarrow [\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau]\!] = [\![\Gamma \vdash e[x := e'] : \tau]\!]$$

- [VAR] *i.e.* for $\Gamma, x : \tau' \vdash v : \tau$. There are two cases:
  - $x = v$ (therefore $\tau = \tau'$) and the semantics:

$$[\text{VAR}]\frac{}{[\![\Gamma, x : \tau' \vdash v : \tau]\!] = \pi_2 \circ \varepsilon}$$

  By the definition of substitution $x[x := e'] = e'$ therefore proof of (*let-β*) for [VAR] where $v = x$ is then:

$$\begin{aligned}
& [\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ x : \tau]\!] \\
= \ & \pi_2 \circ \varepsilon \circ \langle \varepsilon, g \rangle^\dagger \\
= \ & \pi_2 \circ \langle \varepsilon, g \rangle && [\text{coK2}] \\
= \ & g && \{\times \textit{ universality}\} \\
= \ & [\![\Gamma \vdash e' : \tau]\!]
\end{aligned}$$

  - $x \neq v$: thus the semantics is as follows where $|\Gamma| = n$ and v is the i^{th} variable in Γ:

$$[\text{VAR}]\frac{}{[\![\Gamma, x : \tau' \vdash v : \tau]\!] = \pi_2 \circ \pi_1^{(n-i+1)} \circ \varepsilon}$$

  By the definition of substitution $v[x := e'] = v$ therefore proof of (*let-β*) for [VAR] where $v \neq x$ is then:

$$\begin{aligned}
& [\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ x : \tau]\!] \\
= \ & \pi_2 \circ \pi_1^{(n-i+1)} \circ \varepsilon \langle \varepsilon, g \rangle^\dagger \\
= \ & \pi_2 \circ \pi_1^{(n-i+1)} \circ \langle \varepsilon, g \rangle && [\text{coK2}] \\
= \ & \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon && \{\times \textit{ universality}\} \\
= \ & [\![\Gamma \vdash v : \tau]\!]
\end{aligned} \tag{94}$$

- [ABS]

$$[\text{ABS}]\frac{[\![\Gamma, x : \tau', v : \sigma \vdash e : \tau]\!] = f : \mathsf{D}((\Gamma \times \tau') \times \sigma) \to \tau}{[\![\Gamma, x : \tau' \vdash \lambda v.e : \sigma \Rightarrow \tau]\!] = \phi(f \circ \mathsf{m}) : \mathsf{D}(\Gamma \times \tau') \to (\mathsf{D}\sigma \Rightarrow \tau)}$$

By the definition of substitution, $(\lambda v.e)[x := e'] = \lambda v.e[x := e']$, the inductive hypothesis is therefore:

$$(A) =_{[\text{WEAK}]} \frac{[\![\Gamma \vdash e' : \tau']\!] = g : \mathsf{D}\Gamma \to \tau'}{[\![\Gamma, v : \sigma \vdash e' : \tau']\!] = g \circ \mathsf{D}\pi_1 : \mathsf{D}(\Gamma \times \sigma) \to \tau'}$$

$$\frac{\dfrac{[\text{EXCH}]\dfrac{[\![\Gamma, x : \tau', v : \sigma \vdash e : \tau]\!] = f : \mathsf{D}((\Gamma \times \tau') \times \sigma) \to \tau}{[\![\Gamma, v : \sigma, x : \tau' \vdash e : \tau]\!] = f \circ \mathsf{D}\chi : \mathsf{D}((\Gamma \times \sigma) \times \tau') \to \tau} \quad (A)}{\begin{array}{c}[\![\Gamma, v : \sigma \vdash \textbf{let } x = e' \textbf{ in } e : \tau]\!] = (f \circ \mathsf{D}\chi) \circ \langle \varepsilon, g \circ \mathsf{D}\pi_1 \rangle^\dagger : \mathsf{D}(\Gamma \times \sigma) \to \tau \\ = [\![\Gamma, v : \sigma \vdash e[x := e'] : \tau]\!] = h : \mathsf{D}(\Gamma \times \sigma) \to \tau\end{array}} \tag{95}$$

i.e.. $h = (f \circ \mathsf{D}\chi) \circ \langle \varepsilon, g \circ \mathsf{D}\pi_1 \rangle^\dagger$.

The conclusion of (*let-β*) for [ABS] is thus:

$$[\![\Gamma \vdash \textbf{let } x = e' \textbf{ in } \lambda v.e : \sigma \Rightarrow \tau]\!] = \phi(f \circ \mathsf{m}) \circ \langle \varepsilon, g \rangle^\dagger : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau) \tag{96}$$

$$= \quad [\text{ABS}]\frac{[\![\Gamma, v : \sigma \vdash e[x := e'] : \tau]\!] = h : \mathsf{D}(\Gamma \times \sigma) \to \tau}{[\![\Gamma \vdash \lambda v.e[x := e'] : \sigma \Rightarrow \tau]\!] = \phi(h \circ \mathsf{m}) : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau)} \tag{97}$$

Recall that $\chi = \langle \pi_1 \times id, \pi_2 \circ \pi_1 \rangle : (A \times B) \times C \to (A \times C) \times B$ is natural transformation, with naturality $\chi \circ (f \times g) \times h = (f \times h) \times g \circ \chi$ from Example A.3.3 (p. 183).

$$\begin{array}{lll}
& [\![\Gamma \vdash \lambda v.e[x := e'] : \sigma \to \tau]\!] & \\
= & \phi(h \circ \mathsf{m}) & \{(97) \textit{ and } (95)\} \\
= & \phi(f \circ \mathsf{D}\chi \circ \langle \varepsilon, g \circ \mathsf{D}\pi_1 \rangle^\dagger \circ \mathsf{m}) & \{\textit{def.}\} \\
= & \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}\langle \varepsilon, (g \circ \mathsf{D}\pi_1) \rangle \circ \delta \circ \mathsf{m}) & \{f^\dagger = \mathsf{D}f \circ \delta\} \\
= & \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}\langle \varepsilon, (g \circ \mathsf{D}\pi_1) \rangle \circ \mathsf{D}\mathsf{m} \circ \mathsf{m} \circ (\delta \times \delta)) & \{\delta/\mathsf{m} \textit{ lax monoidal comonad}\} \\
= & \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}\langle (\varepsilon \circ \mathsf{m}), (g \circ \mathsf{D}\pi_1 \circ \mathsf{m}) \rangle \circ \mathsf{m} \circ (\delta \times \delta)) & \{\langle f, g \rangle \circ h = \langle f \circ h, g \circ h \rangle\} \\
= & \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}\langle (\varepsilon \times \varepsilon), (g \circ \mathsf{D}\pi_1 \circ \mathsf{m}) \rangle \circ \mathsf{m} \circ (\delta \times \delta)) & \{\epsilon/\mathsf{m} \textit{ lax monoidal comonad}\} \\
= & \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}((\varepsilon \times \varepsilon) \times (g \circ \mathsf{D}\pi_1 \circ \mathsf{m})) \circ \mathsf{D}\Delta \circ \mathsf{m} \circ (\delta \times \delta)) & \{\langle f, g \rangle = (f \times g) \circ \Delta\} \\
= & \phi(f \circ \mathsf{D}((\varepsilon \times (g \circ \mathsf{D}\pi_1 \circ \mathsf{m})) \times \varepsilon) \circ \mathsf{D}\chi \circ \mathsf{D}\Delta \circ \mathsf{m} \circ (\delta \times \delta)) & \{\chi \textit{ naturality}\} \\
= & \phi(f \circ \mathsf{D}\langle (\varepsilon \times (g \circ \mathsf{D}\pi_1 \circ \mathsf{m})) \circ (\pi_1 \times id), \varepsilon \circ \pi_2 \circ \pi_1 \rangle \circ \mathsf{D}\Delta \circ \mathsf{m} \circ (\delta \times \delta)) & \{\chi \textit{ defn. \& } \times \textit{ universality}\} \\
= & \phi(f \circ \mathsf{D}\langle (\varepsilon \circ \pi_1) \times (g \circ \mathsf{D}\pi_1 \circ \mathsf{m}), \varepsilon \circ \pi_2 \circ \pi_1 \rangle \circ \mathsf{D}\Delta \circ \mathsf{m} \circ (\delta \times \delta)) & \{\times \textit{ bifunctor}\} \\
= & \phi(f \circ \mathsf{D}\langle ((\varepsilon \circ \pi_1) \times (g \circ \mathsf{D}\pi_1 \circ \mathsf{m})) \circ \Delta, \varepsilon \circ \pi_2 \circ \pi_1 \circ \Delta \rangle \circ \mathsf{m} \circ (\delta \times \delta)) & \{\times \textit{ universal prop.}\} \\
= & \phi(f \circ \mathsf{D}\langle ((\varepsilon \circ \pi_1) \times (g \circ \mathsf{D}\pi_1 \circ \mathsf{m})) \circ \Delta, \varepsilon \circ \pi_2 \rangle \circ \mathsf{m} \circ (\delta \times \delta)) & \{\pi_1 \circ \Delta = id\} \\
= & \phi(f \circ \mathsf{D}\langle ((\varepsilon \circ \pi_1) \times (g \circ \pi_1)) \circ \Delta, \varepsilon \circ \pi_2 \rangle \circ \mathsf{m} \circ (\delta \times \delta)) & \{\mathsf{D}\pi_1 \circ \mathsf{m} = \pi_1\} \\
= & \phi(f \circ \mathsf{D}\langle (\varepsilon \times g) \circ \Delta \circ \pi_1, \varepsilon \circ \pi_2 \rangle \circ \mathsf{m} \circ (\delta \times \delta)) & \{\Delta \textit{ naturality}\} \\
= & \phi(f \circ \mathsf{D}(((\varepsilon \times g) \circ \Delta) \times \varepsilon) \circ \mathsf{m} \circ (\delta \times \delta)) & \{(\pi_1 \times \pi_2) \circ \Delta = id\} \\
= & \phi(f \circ \mathsf{m} \circ (\mathsf{D}((\varepsilon \times g) \circ \Delta) \times \mathsf{D}\varepsilon) \circ (\delta \times \delta)) & \{\mathsf{m} \textit{ naturality}\} \\
= & \phi(f \circ \mathsf{m} \circ (\mathsf{D}((\varepsilon \times g) \circ \Delta) \circ \delta \times \mathsf{D}\varepsilon \circ \delta)) & \{\times \textit{ bifunctor}\} \\
= & \phi(f \circ \mathsf{m} \circ (\mathsf{D}((\varepsilon \times g) \circ \Delta) \circ \delta \times id)) & \{[\text{C2}] \textit{ comonad law}\} \\
= & \phi(f \circ \mathsf{m} \circ (\langle \varepsilon, g \rangle^\dagger \times id)) & \{f^\dagger = \mathsf{D}f \circ \delta \textit{ and } \times \textit{ universality}\} \\
= & \phi(f \circ \mathsf{m}) \circ \langle \varepsilon, g \rangle^\dagger & \{\textit{adjunctions}\} \\
= & [\![\Gamma \vdash \textbf{let } e' = x \textbf{ in } \lambda v.e : \sigma \to \tau]\!] & \{\textit{defn.}\} \quad \square
\end{array}$$

▸ [APP]

$$[\text{APP}]\frac{[\![\Gamma, x:\tau' \vdash e_1 : \sigma \Rightarrow \tau]\!] = k_1 : \mathsf{D}(\Gamma \times \tau') \to (\mathsf{D}\sigma \Rightarrow \tau) \qquad [\![\Gamma, x:\tau' \vdash e_2 : \sigma]\!] = k_2 : \mathsf{D}\Gamma \to \sigma}{[\![\Gamma, x:\tau' \vdash e_1\, e_2 : \tau]\!] = \phi^{-1} k_1 \circ \mathsf{n}_{\Gamma\times\tau',\sigma} \circ \langle \varepsilon, k_2 \rangle^\dagger : \mathsf{D}(\Gamma \times \tau') \to \tau}$$

By the definition of substitution $(e_1\, e_2)[x := e'] = e_1[x := e']\, e_2[x := e']$. Inductive hypotheses are therefore:

$$\begin{aligned} &[\![\Gamma \vdash e_1[x := e'] : \sigma \Rightarrow \tau]\!] = h_1 : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau \\ = \quad &[\text{LET}]\frac{[\![\Gamma, x:\tau' \vdash e_1 : \sigma \Rightarrow \tau]\!] = k_1 : \mathsf{D}(\Gamma \times \tau') \to (\mathsf{D}\sigma \Rightarrow \tau) \qquad [\![\Gamma \vdash e' : \tau']\!] = g : \mathsf{D}\Gamma \to \tau'}{[\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e_1 : \sigma \Rightarrow \tau]\!] = k_1 \circ \langle \varepsilon, g \rangle^\dagger : \mathsf{D}\Gamma \to (\mathsf{D}\sigma \Rightarrow \tau)} \end{aligned} \tag{98}$$

$$\begin{aligned} &[\![\Gamma \vdash e_2[x := e'] : \sigma]\!] = h_2 : \mathsf{D}\Gamma \to \sigma \\ = \quad &[\text{LET}]\frac{[\![\Gamma, x:\tau' \vdash e_2 : \sigma]\!] = k_2 : \mathsf{D}(\Gamma \times \tau') \to \sigma \qquad [\![\Gamma \vdash e' : \tau']\!] = g : \mathsf{D}\Gamma \to \tau'}{[\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e_2 : \sigma]\!] = k_2 \circ \langle \varepsilon, g \rangle^\dagger : \mathsf{D}\Gamma \to \sigma} \end{aligned} \tag{99}$$

The conclusion of (*let-β*) for [APP] is thus:

$$\begin{aligned} &[\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e_1\, e_2 : \tau]\!] = \phi^{-1} k_1 \circ \mathsf{n}_{\Gamma,\sigma} \circ \langle \varepsilon, k_2 \rangle^\dagger \circ \langle \varepsilon, g \rangle^\dagger : \mathsf{D}\Gamma \to \tau \\ = &[\![\Gamma \vdash e_1[x := e']\, e_2[x := e'] : \tau]\!] = \phi^{-1} h_1 \circ \mathsf{n}_{\Gamma,\sigma} \circ \langle \varepsilon, h_2 \rangle^\dagger \end{aligned} \tag{100}$$

$$\begin{aligned}
& [\![\Gamma \vdash (e_1\, e_2)[x := e'] : \tau]\!] \\
= \quad & \phi^{-1}(k_1 \circ \langle \varepsilon, g \rangle^\dagger) \circ \mathsf{n} \circ \langle \varepsilon, k_2 \circ \langle \varepsilon, g \rangle^\dagger \rangle^\dagger && \{(100),\ \textit{and}\ (98),(99)\} \\
= \quad & \phi^{-1} k_1 \circ (\langle \varepsilon, g \rangle^\dagger \times \mathit{id}) \circ \mathsf{n} \circ \langle \varepsilon, k_2 \circ \langle \varepsilon, g \rangle^\dagger \rangle^\dagger && \{\phi^{-1}(g \circ f) = \phi^{-1} g \circ (- \times X) f\} \\
= \quad & \phi^{-1} k_1 \circ (\langle \varepsilon, g \rangle^\dagger \times \mathit{id}) \circ \mathsf{n} \circ \mathsf{D}((\varepsilon \times (k_2 \circ \langle \varepsilon, g \rangle^\dagger)) \circ \Delta) \circ \delta && \{f^\dagger = \mathsf{D}f \circ \delta\ \textit{and}\ \langle g, f \rangle = (g \times f) \circ \Delta\} \\
= \quad & \phi^{-1} k_1 \circ (\langle \varepsilon, g \rangle^\dagger \times \mathit{id}) \circ ((\mathsf{D}\varepsilon) \times \mathsf{D}(k_2 \circ \langle \varepsilon, g \rangle^\dagger)) \circ \mathsf{n} \circ \mathsf{D}\Delta \circ \delta && \{\mathsf{D}\ \textit{functoriality and}\ \mathsf{n}\ \textit{naturality}\} \\
= \quad & \phi^{-1} k_1 \circ (\langle \varepsilon, g \rangle^\dagger \times \mathit{id}) \circ ((\mathsf{D}\varepsilon) \times \mathsf{D}(k_2 \circ \langle \varepsilon, g \rangle^\dagger)) \circ (\delta \times \delta) \circ \mathsf{n} \circ \mathsf{D}\Delta && \{\mathsf{n} \circ \mathsf{D}\Delta\ \textit{naturality}\} \\
= \quad & \phi^{-1} k_1 \circ ((\langle \varepsilon, g \rangle^\dagger \circ (\mathsf{D}\varepsilon) \circ \delta) \times (\mathsf{D}(k_2 \circ \langle \varepsilon, g \rangle^\dagger) \circ \delta)) \circ \mathsf{n} \circ \mathsf{D}\Delta && \{\times\ \textit{functoriality}\} \\
= \quad & \phi^{-1} k_1 \circ (\langle \varepsilon, g \rangle^\dagger \times (\mathsf{D}(k_2 \circ \langle \varepsilon, g \rangle^\dagger) \circ \delta)) \circ \mathsf{n} \circ \mathsf{D}\Delta && \{\mathsf{D}\epsilon \circ \delta = \mathit{id}\ [\text{C2}]\} \\
= \quad & \phi^{-1} k_1 \circ ((\mathsf{D}\langle \varepsilon, g \rangle \circ \delta) \times (\mathsf{D}(k_2 \circ \langle \varepsilon, g \rangle^\dagger) \circ \delta)) \circ \mathsf{n} \circ \mathsf{D}\Delta && \{f^\dagger = \mathsf{D}f \circ \delta\} \\
= \quad & \phi^{-1} k_1 \circ (\mathsf{D}\langle \varepsilon, g \rangle \times (\mathsf{D}(k_2 \circ \langle \varepsilon, g \rangle^\dagger))) \circ (\delta \times \delta) \circ \mathsf{n} \circ \mathsf{D}\Delta && \{\times\ \textit{functoriality}\} \\
= \quad & \phi^{-1} k_1 \circ (\mathsf{D}\langle \varepsilon, g \rangle \times (\mathsf{D}(k_2 \circ \langle \varepsilon, g \rangle^\dagger))) \circ \mathsf{n} \circ \mathsf{D}\mathsf{n} \circ \delta \circ \mathsf{D}\Delta && \{\delta/\mathsf{n}\ \textit{colax monoidal comonad}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}(\langle \varepsilon, g \rangle \times (k_2 \circ \langle \varepsilon, g \rangle^\dagger)) \circ \mathsf{D}\mathsf{n} \circ \delta \circ \mathsf{D}\Delta && \{\mathsf{n}\ \textit{naturality}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}(\langle \varepsilon, g \rangle \times (k_2 \circ \langle \varepsilon, g \rangle^\dagger)) \circ \mathsf{D}\mathsf{n} \circ \mathsf{D}\mathsf{D}\Delta \circ \delta && \{\delta\ \textit{naturality}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}(\langle \varepsilon, g \rangle \times (k_2 \circ \langle \varepsilon, g \rangle^\dagger)) \circ \mathsf{D}\Delta \circ \delta && \{\mathsf{n}\ \textit{idempotence}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}((\langle \varepsilon, g \rangle \circ \varepsilon \circ \delta) \times (k_2 \circ \mathsf{D}\langle \varepsilon, g \rangle \circ \delta)) \circ \mathsf{D}\Delta \circ \delta && \{\varepsilon \circ \delta = \mathit{id}\ \textit{[C1]}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}((\varepsilon \circ \mathsf{D}\langle \varepsilon, g \rangle \circ \delta) \times (k_2 \circ \mathsf{D}\langle \varepsilon, g \rangle \circ \delta)) \circ \mathsf{D}\Delta \circ \delta && \{\varepsilon\ \textit{naturality}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}(\varepsilon \times k_2) \circ \mathsf{D}\Delta \circ \mathsf{D}(\mathsf{D}\langle \varepsilon, g \rangle \circ \delta) \circ \delta && \{\Delta\ \textit{naturality},\ \times\ \textit{functoriality}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}(\varepsilon \times k_2) \circ \mathsf{D}\Delta \circ \mathsf{D}\mathsf{D}\langle \varepsilon, g \rangle \circ \delta \circ \delta && \{\mathsf{D}\delta \circ \delta = \delta_{\mathsf{D}} \circ \delta\ [\text{C3}]\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \mathsf{D}(\varepsilon \times k_2) \circ \mathsf{D}\Delta \circ \delta \circ \mathsf{D}\langle \varepsilon, g \rangle \circ \delta && \{\delta\ \textit{naturality}\} \\
= \quad & \phi^{-1} k_1 \circ \mathsf{n} \circ \langle \varepsilon, k_2 \rangle^\dagger \circ \langle \varepsilon, g \rangle^\dagger && \{f^\dagger = \mathsf{D}f \circ \delta\ \textit{and}\ g \times f \circ \Delta = \langle g, f \rangle\} \\
= \quad & [\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e_1\, e_2 : \tau]\!] && (100)
\end{aligned}$$

C.2.4. Monoidal comonads and shape

Theorem C.2.5. *For an idempotent monoidal functor* F*, the canonical colax monoidal operation* $\mathsf{n} = \langle \mathsf{D}\pi_1, \mathsf{D}\pi_2 \rangle$ *is the left-inverse of the monoidal operation* m *when pre-composed with morphisms whose images are of equal shape i.e. for* $f : A \to \mathsf{F}X, y : B \to \mathsf{F}Y$*, then:*

$$shape_X \circ f \circ \pi_1 = shape_Y \circ g \circ \pi_2 \quad \Rightarrow \quad \mathsf{n}_{X,Y} \circ \mathsf{m}_{X,Y} \circ (f \times g) = f \times g \tag{101}$$

Proof. The following proof makes use of the following property derived from the universal property of products, where for all $f : A \times X, g : B \times Y$:

$$f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle \tag{102}$$

The proof is then as follows (where some of the steps will be applied to both sides of a product):

$$\begin{aligned}
& \mathsf{n} \circ \mathsf{m} \circ (f \times g) \\
= \; & \langle \mathsf{D}\pi_1, \mathsf{D}\pi_2 \rangle \circ \mathsf{m} \circ (f \times g) && \{\textit{canonical } \mathsf{n}\} \\
= \; & \langle \mathsf{D}(\pi_1 \circ (id \times !_Y)), \mathsf{D}(\pi_2 \circ (!_X \times id)) \rangle \circ \mathsf{m} \circ (f \times g) && \{\pi_1 = \pi_1 \circ (id \times f)\} \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{D}(id \times !_Y) \circ \mathsf{m} \circ (f \times g), \mathsf{D}\pi_2 \circ \mathsf{D}(!_X \times id) \circ \mathsf{m} \circ (f \times g) \rangle && \{\times \textit{ universality}\} \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{m} \circ (\mathsf{D}id \times \mathsf{D}!_Y) \circ (f \times g), \mathsf{D}\pi_2 \circ \mathsf{m} \circ (\mathsf{D}!_X \times \mathsf{D}id) \circ (f \times g) \rangle && \{\mathsf{m} \textit{ naturality}\} \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{m} \circ ((\mathsf{D}id \circ f) \times (\mathsf{D}!_Y \circ g)), \mathsf{D}\pi_2 \circ \mathsf{m} \circ ((\mathsf{D}!_X \circ f) \times (\mathsf{D}id \circ g)) \rangle && \{\times \textit{ functoriality}\} \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{m} \circ \langle \mathsf{D}id \circ f \circ \pi_1, \mathsf{D}!_Y \circ g \circ \pi_2 \rangle, \mathsf{D}\pi_2 \circ \mathsf{m} \circ \langle \mathsf{D}!_X \circ f \circ \pi_1, \mathsf{D}id \circ g \circ \pi_2 \rangle \rangle && (102) \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{m} \circ \langle \mathsf{D}id \circ f \circ \pi_1, \mathsf{D}!_x \circ f \circ \pi_1 \rangle, \mathsf{D}\pi_2 \circ \mathsf{m} \circ \langle \mathsf{D}!_Y \circ g \circ \pi_2, \mathsf{D}id \circ g \circ \pi_2 \rangle \rangle && \{(101) \textit{ premise}\} \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{m} \circ \langle \mathsf{D}id, \mathsf{D}!_X \rangle \circ f \circ \pi_1, \mathsf{D}\pi_2 \circ \mathsf{m} \circ \langle \mathsf{D}!_Y, \mathsf{D}id \rangle \circ g \circ \pi_2 \rangle && \{\times \textit{ universality}\} \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{D}(id \times !_x) \circ \mathsf{m} \circ \Delta \circ f \circ \pi_1, \mathsf{D}\pi_2 \circ \mathsf{D}(!_Y \times id) \circ \mathsf{m} \circ \Delta \circ g \circ \pi_2 \rangle && \{\mathsf{m} \textit{ naturality}\} \\
= \; & \langle \mathsf{D}\pi_1 \circ \mathsf{D}(id \times !_x) \circ \mathsf{D}\Delta \circ f \circ \pi_1, \mathsf{D}\pi_2 \circ \mathsf{D}(!_Y \times id) \circ \mathsf{D}\Delta \circ g \circ \pi_2 \rangle && \{\mathsf{m} \textit{ idempotency}\} \\
= \; & \langle \mathsf{D}(\pi_1 \circ \langle id, !_x \rangle) \circ f \circ \pi_1, \mathsf{D}(\pi_2 \circ \langle !_Y, id \rangle) \circ g \circ \pi_2 \rangle && \{\times \textit{ universality}\} \\
= \; & \langle f \circ \pi_1, g \circ \pi_2 \rangle && \{\times \textit{ universality}\} \\
= \; & f \times g && (102)
\end{aligned}$$

□

C.3. Generalising comonads

C.3.1. Lattices of comonads

Lemma C.3.1. *Given a bounded semilattice of comonads* $(\mathsf{D}, \sqcup, \bot)$*, there is a category* ${}_{\mathsf{D}}\mathbb{C}$ *with objects* ${}_{\mathsf{D}}\mathbb{C}_0 = \mathbb{C}_0$ *and morphisms* ${}_{\mathsf{D}}\mathbb{C}(A, B) = \bigcup_{X \in \mathbb{I}_0} \mathbb{C}(\mathsf{D}_X A, B)$ *with identities* $id_A : \mathsf{D}_\bot A \to A = (\varepsilon_\bot)_A$ *and composition defined:*

$$\begin{aligned}
\hat{\circ} : (\mathsf{D}_Y B \to C) \to (\mathsf{D}_X A \to B) \to (\mathsf{D}_{(X \sqcup Y)} A \to C) \\
g \,\hat{\circ} f = g \circ \iota_Y^{X \cup Y} \circ (f \circ \iota_X^{X \cup Y})^\dagger
\end{aligned} \tag{103}$$

and identities $id_A : \mathsf{D}_\bot A \to A = (\varepsilon_\bot)_A$.

Proof. Proof of the axioms of a category rely on the comonad morphism properties:

$$\begin{array}{ccccccc} A & \xleftarrow{\varepsilon_A} & \mathsf{D}A & \xrightarrow{\delta_A} & \mathsf{DD}A & \xrightarrow{\iota_{\mathsf{D}A}} & \mathsf{D}'\mathsf{D}A \\ & \nwarrow^{\varepsilon'_A} & \downarrow{\scriptstyle \iota_A} & & & \swarrow_{\mathsf{D}'\iota_A} & \\ & & \mathsf{D}'A & \xrightarrow[\delta'_A]{} & \mathsf{D}'\mathsf{D}'A & & \end{array} \tag{104}$$

Furthermore, transitivity and reflexivity properties of comonad homomorphisms are used:

$$\iota_R^{R\sqcup S\sqcup T} = \iota_R^{S\sqcup T} \circ \iota_{S\sqcup T}^{R\sqcup S\sqcup T} = \iota_R^{R\sqcup S} \circ \iota_{R\sqcup S}^{R\sqcup S\sqcup T} \tag{105}$$

$$\iota_T^S \sqcup \iota_S^R = \iota_T^R \tag{106}$$

$$\iota_R^R = id \tag{107}$$

$$\iota_R^{R\sqcup S} = \iota_R^{S\sqcup R} \tag{108}$$

Proof of the categorical laws are then as follows:

- $\hat{id} \mathbin{\hat{\circ}} f = f$ for all $f : D_R X \to Y$.

$$\begin{array}{rcll} \hat{id} \mathbin{\hat{\circ}} f = & \varepsilon_\perp \circ \iota_\perp^R \circ \mathsf{D}_R(f \circ \iota_R^R) \circ \delta_{R\sqcup\perp} & & \\ = & \varepsilon_R \circ \mathsf{D}_R(f \circ \iota_R^R) \circ \delta_{R\sqcup\perp} & & (104) \\ = & \varepsilon_R \circ \mathsf{D}_R f \circ \delta_{R\sqcup\perp} & & (107) \\ = & f \circ \varepsilon_R \circ \delta_{R\sqcup\perp} & & \{\varepsilon_R \textit{ naturality}\} \\ = & f \circ \varepsilon_R \circ \delta_R & & \{R \sqcup \perp = R\} \\ = & f & & \{[\mathrm{C1}] \textit{ for } \mathsf{D}_R\} \end{array}$$

- $f \mathbin{\hat{\circ}} \hat{id} = f$ for all $f : D_R X \to Y$.

$$\begin{array}{rcll} f \mathbin{\hat{\circ}} \hat{id} = & (f \circ \iota_R^R) \circ \mathsf{D}_R(\varepsilon_\perp \circ \iota_\perp^R) \circ \delta_{R\sqcup\perp} & & \\ = & (f \circ \iota_R^R) \circ \mathsf{D}_R(\varepsilon_R) \circ \delta_{R\sqcup\perp} & & (104) \\ = & (f \circ \iota_R^R) \circ \mathsf{D}_R(\varepsilon_R) \circ \delta_R & & \{R \sqcup \perp = R\} \\ = & (f \circ \iota_R^R) & & \{[\mathrm{C1}] \textit{ for } \mathsf{D}_R\} \\ = & f & & (107) \end{array}$$

- $h \mathbin{\hat{\circ}} (g \mathbin{\hat{\circ}} f) = (h \mathbin{\hat{\circ}} g) \mathbin{\hat{\circ}} f$ for all $f : \mathsf{D}_r X \to Y, g : \mathsf{D}_s Y \to Z, h : \mathsf{D}_t Z \to W$:

$$\begin{array}{rl} & h \mathbin{\hat{\circ}} (g \mathbin{\hat{\circ}} f) \\ = & h \mathbin{\hat{\circ}} (g \circ \iota_s^{r\vee s} \circ \mathsf{D}_{r\vee s}(f \circ \iota_r^{r\vee s}) \circ \delta_{r\vee s}) \\ = & h \circ \iota_t^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(g \circ \iota_s^{r\vee s} \circ \mathsf{D}_{r\vee s}(f \circ \iota_r^{r\vee s}) \circ \delta_{r\vee s} \circ \iota_{r\vee s}^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(g \circ \iota_s^{r\vee s} \circ \mathsf{D}_{r\vee s}(f \circ \iota_r^{r\vee s}) \circ \delta_{r\vee s} \circ \iota_{r\vee s}^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\mathsf{D}_{r\vee s}(f \circ \iota_r^{r\vee s}) \circ \delta_{r\vee s} \circ \iota_{r\vee s}^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\mathsf{D}_{r\vee s}(f \circ \iota_r^{r\vee s}) \circ \mathsf{D}_{r\vee s}(\iota_{r\vee s}^{r\vee s\vee t}) \circ \iota_{r\vee s}^{r\vee s\vee t} \circ \delta_{r\vee s\vee r}) \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\mathsf{D}_{r\vee s}(f \circ \iota_r^{r\vee s\vee t}) \circ \iota_{r\vee s}^{r\vee s\vee t} \circ \delta_{r\vee s\vee r}) \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\iota_{r\vee s}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ \delta_{r\vee s\vee r}) \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\iota_{r\vee s}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t})) \circ \mathsf{D}_{r\vee s\vee t}\delta_{r\vee s\vee t} \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\iota_{r\vee s}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t})) \circ \delta_{r\vee s\vee t} \circ \delta_{r\vee s\vee t} \\ = & h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\iota_{r\vee s}^{r\vee s\vee t}) \circ \mathsf{D}_{r\vee s\vee t}\mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \circ \delta_{r\vee s\vee t} \end{array}$$

$$
\begin{aligned}
&= h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(\iota_{r\vee s}^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\
&= h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s}) \circ \mathsf{D}_{s\vee t}(\iota_{r\vee s}^{r\vee s\vee t}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \delta_{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\
&= h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{r\vee s} \circ \iota_{r\vee s}^{r\vee s\vee t}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \delta_{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\
&= h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{s\vee t} \circ \iota_{s\vee t}^{r\vee s\vee t}) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \delta_{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\
&= h \circ \iota_t^{s\vee t} \circ \mathsf{D}_{s\vee t}(g \circ \iota_s^{s\vee t}) \circ \delta_{s\vee t} \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ \delta_{r\vee s\vee t} \\
&= (h \mathbin{\hat{\circ}} g) \circ \iota_{s\vee t}^{r\vee s\vee t} \circ \mathsf{D}_{r\vee s\vee t}(f \circ \iota_r^{r\vee s\vee t}) \circ_{r\vee s\vee t} \\
&= (h \mathbin{\hat{\circ}} g) \mathbin{\hat{\circ}} f
\end{aligned}
$$

□

C.4. The coeffect calculus

C.4.1. Categorical semantics

C.4.1.1. Monoidal category from coeffect algebra

Proposition C.4.1. *For a coeffect algebra $(C, \bullet, I, \sqcup, \sqcap)$ where $\mathbb{I}$ is the category of the partial order $(C, \sqsubseteq)$, then $(\mathbb{I}^{\mathsf{op}}, \bullet, I)$ is a* monoidal category *with bifunctor $\bullet : \mathbb{I}^{\mathsf{op}} \times \mathbb{I}^{\mathsf{op}} \to \mathbb{I}^{\mathsf{op}}$.*

Proof. Bifunctorality of $\bullet$ means that given $f : X \to Y, g : A \to B \in \mathbb{I}_1^{\mathsf{op}}$ then $f \bullet g : X \bullet A \to Y \bullet B \in \mathbb{I}_1^{\mathsf{op}}$ and therefore $Y \sqsubseteq X$ and $B \sqsubseteq A$ implies $(Y \bullet B) \sqsubseteq (X \bullet A)$.

$$
\begin{array}{lllll}
(1) & Y \sqsubseteq X \Rightarrow & Y \sqcup X & = X & \\
 & & (Y \sqcup X) \bullet A & = X \bullet A & \{- \bullet A\} \\
 & & (Y \bullet A) \sqcup (X \bullet A) & = X \bullet A & \{\textit{distributivity}\} \\
(2) & B \sqsubseteq A \Rightarrow & B \sqcup A & = A & \\
 & & X \bullet (B \sqcup A) & = X \bullet A & \{X \bullet -\} \\
 & & (X \bullet B) \sqcup (X \bullet A) & = X \bullet A & \{\textit{distributivity}\}
\end{array}
$$

$$
\begin{aligned}
X \bullet A &= (Y \sqcup X) \bullet (B \sqcup A) \\
&= (Y \bullet (B \sqcup A)) \sqcup (X \bullet (B \sqcup A)) && \{\textit{distributivity}\} \\
&= (Y \bullet B) \sqcup (Y \bullet A) \sqcup (X \bullet B) \sqcup (X \bullet A) && \{\textit{distributivity and} \sqcup \textit{associativity}\} \\
&= (Y \bullet B) \sqcup (Y \bullet A) \sqcup (X \bullet A) && (2) \\
X \bullet A &= (Y \bullet B) \sqcup (X \bullet A) && (1) \\
\Rightarrow\ & (Y \bullet B) \sqsubseteq (X \bullet A)
\end{aligned}
$$

□

C.4.2. Equational theory - syntactically

Lemma 6.3.2. (p. 124) (Substitution) Given a *substituting* coeffect algebra over C (see Definition 6.3.3, p. 124) $\Gamma, x : \tau' \,?\, R \vdash e : \tau$, and $\Gamma \,?\, S \vdash e' : \tau'$ then $\Gamma \,?\, R \sqcup (R \bullet S) \vdash e[x := e'] : \tau$.

Proof. By induction on typing/coeffect rules:

▸ [VAR] Base case, with coeffect/typing:

$$[\text{VAR}]\frac{(v : \tau) \in (\Gamma, x : \tau')}{(\Gamma, x : \tau') \,?\, I \vdash v : \tau}$$

Therefore, substitution should yield a coeffect $I \sqcup (I \bullet X)$.

By the definition of substitution (10) there are two possible cases for $v[x := e']$:

1. $v = x \therefore x[x := e'] = e'$ and $\tau = \tau'$, and therefore $\Gamma \,?\, X \vdash x[x := e'] : \tau'$.
   From the premise of the lemma, I is either the lower-bound or upper-bound of the (semi)lattice $(C, \sqcup)$ therefore:
   - $\forall X.I \sqsubseteq X$ (lower-bound) then $I \sqcup X = X$ therefore: $X = I \sqcup (I \bullet X)$.
   - $\forall X.X \sqsubseteq I$ (upper-bound) then:

     $$[\text{SUB}]\ \frac{\Gamma \,?\, X \vdash e' : \tau' \qquad X \sqsubseteq I}{\Gamma \,?\, I \vdash e' : \tau'}$$

     where $I = I \sqcup (I \bullet X)$.
2. $v \neq x \therefore v[x := e'] = v$, therefore $\Gamma \,?\, I \vdash v[x := e'] : \tau$.
   From the premise of the lemma, I is either the lower-bound or upper-bound of the (semi)lattice $(C, \sqcup)$ therefore:
   - $\forall X.I \sqsubseteq X$ (lower-bound) then $I \sqcup X = X$ therefore:

     $$\begin{array}{l}[\text{SUB}]\ \dfrac{\Gamma \,?\, I \vdash v : \tau \qquad I \sqsubseteq X}{\Gamma \,?\, X \vdash v : \tau} \\ [\cong]\ \dfrac{}{\Gamma \,?\, I \sqcup (I \bullet X) \vdash v : \tau}\end{array}$$

   - $\forall X.X \sqsubseteq I$ (upper-bound) then $X \sqcup I = I$, therefore: $I = I \sqcup (I \bullet X)$.

▸ [APP] – Application has coeffects/typing:

$$[\text{APP}]\frac{\Gamma, x : \tau' \,?\, R \vdash e_1 : \sigma \xrightarrow{S} \tau \qquad \Gamma, x : \tau' \,?\, T \vdash e_2 : \sigma}{\Gamma, x : \tau' \,?\, R \sqcup (S \bullet T) \vdash e_1\ e_2 : \tau}$$

By the inductive hypothesis:

$$\Gamma \,?\, R \sqcup (R \bullet X) \vdash e_1[x := e'] : \sigma \xrightarrow{S} \tau \tag{109}$$

$$\Gamma \,?\, T \sqcup (T \bullet X) \vdash e_2[x := e'] : \sigma \tag{110}$$

Therefore, from the definition of substitution $(e_1\ e_2)[x := e'] = e_1[x := e']\, e_2[x := e']$:

$$[\text{APP}]\frac{\Gamma \,?\, R \sqcup (R \bullet X) \vdash e_1 : \sigma \xrightarrow{S} \tau \qquad \Gamma \,?\, T \sqcup (T \bullet X) \vdash e_2 : \sigma}{\Gamma \,?\, (R \sqcup (R \bullet X)) \sqcup (S \bullet (T \sqcup (T \bullet X))) \vdash e_1[x := e']\, e_2[x := e'] : \tau}$$

$$\begin{array}{lll} & (R \sqcup (R \bullet X)) \sqcup (S \bullet (T \sqcup (T \bullet X))) & \\ = & R \sqcup (R \bullet X) \sqcup (S \bullet T) \sqcup (S \bullet T \bullet X) & \{\sqcup \textit{ associativity and } \bullet/\sqcup \textit{ distributivity}\} \\ = & R \sqcup (S \bullet T) \sqcup ((R \sqcup (S \bullet T)) \bullet X)\ \square & \{\bullet/\sqcup \textit{ distributivity}\} \end{array}$$

▸ [ABS] – Abstraction has coeffects/typing

$$[\text{ABS}]\frac{\Gamma, x : \tau', v : \sigma \,?\, S \sqcap T \vdash e : \tau}{\Gamma, x : \tau' \,?\, S \vdash \lambda v.e : \sigma \xrightarrow{T} \tau}$$

By the inductive hypothesis:

$$\Gamma, v : \sigma \,?\, (S \sqcap T) \sqcup ((S \sqcap T) \bullet X) \vdash e : \tau \tag{111}$$

Therefore, following from the definition of substitution $\lambda v.e[x := e'] = \lambda v.e[x := e']$ (assuming α-renaming such that $x \neq v$):

$$[\text{ABS}]\frac{\Gamma, v : \sigma \,?\, (S \sqcap T) \sqcup ((S \sqcap T) \bullet X) \vdash e : \tau}{\Gamma \,?\, P \vdash \lambda v.e[x := e'] : \sigma \xrightarrow{Q} \tau} \text{ where } P \sqcap Q = (S \sqcap T) \sqcup ((S \sqcap T) \bullet X)$$

where for substitution to hold then $P = S \sqcup (S \bullet X)$ and $Q = T$. Therefore it is required that $(S \sqcup (S \bullet X)) \sqcap T = (S \sqcap T) \sqcup ((S \sqcap T) \bullet X)$. This condition holds if $\bullet$ distributes over $\sqcup$ as in the [APP] rule, and given an interchange law between $\sqcap$ and $\bullet$:

$$(X \bullet Y) \sqcap (A \bullet B) = (X \sqcap A) \bullet (Y \sqcap B) \tag{112}$$

and if $(C, \sqcap, I)$ is a monoid:

$$\begin{array}{lll}
 & (S \sqcup (S \bullet X)) \sqcap T & \\
= & (S \sqcup (S \bullet X)) \sqcap (T \bullet I) & \{I/\bullet \textit{ monoid}\} \\
= & ((S \bullet I) \sqcup (S \bullet X)) \sqcap (T \bullet I) & \{I/\bullet \textit{ monoid}\} \\
= & (S \bullet (I \sqcup X)) \sqcap (T \bullet I) & \{\bullet \textit{ distrib over } \sqcup\} \\
= & (S \sqcap T) \bullet ((I \sqcup X) \sqcap I) & \{\bullet/\sqcap \textit{ interchange (112)}\} \\
= & (S \sqcap T) \bullet (I \sqcup X) & \{I/\sqcap \textit{ monoid}\} \\
= & ((S \sqcap T) \bullet I) \sqcup ((S \sqcap T) \bullet X) & \{\bullet/\sqcup \textit{ distributivity}\} \\
= & (S \sqcap T) \sqcup ((S \sqcap T) \bullet X) \; \square & \{I/\bullet \textit{ monoid}\}
\end{array}$$

□

C.4.3. Equational theory - categorical semantics

In this section, to reduce the notational clutter, the indexed monoidal functors operations $\mathsf{m}^{R,S}_{A,B}$ and $\mathsf{n}^{R,S}_{A,B}$ are written $\mathsf{m}_{R,S}$ and $\mathsf{n}_{R,S}$ (*i.e.*, the non-index objects are elided), and m^{I}_{A} and n^{I}_{A} are written m_{I} and n_{I} respectively.

Let-binding and abstraction-application equality.

Proposition C.4.2 ([*let*-λ-$\equiv$])**.**

$$[\![\Gamma \,?\, X \sqcup (Y \bullet Z) \vdash (\lambda v.e_1)\, e_2 : \tau]\!] \equiv [\![\Gamma \,?\, X \sqcup (Y \bullet Z) \vdash \textbf{\textit{let}}\, x = e_1 \;\textbf{\textit{in}}\, e_2 : \tau]\!]$$

if $\delta_{R,S}$ is colax monoidal (therefore $\bullet$ and $\sqcup$ have interchange), and if $\mathsf{m}_{R,S} = \mathsf{n}_{R,S} = id$ (therefore $\sqcup = \sqcap$), and $\sqcup$ is idempotent.

Proof.

$$
\begin{array}{lll}
& [\![\Gamma \,?\, X \sqcup (Y \bullet Z) \vdash (\lambda v.e_1)\, e_2 : \tau]\!] & \\
= & \psi(\phi(k_1 \circ \mathsf{m}_{X,Y})) \circ (id \times \mathsf{D}k_2 \circ \delta_{Y,Z}) \circ \mathsf{n}_{X,Y \bullet Z} \circ \mathsf{D}\Delta & \\
= & k_1 \circ \mathsf{m}_{X,Y} \circ (id \times \mathsf{D}k_2 \circ \delta_{X,Y}) \circ \mathsf{n}_{X,Y \bullet Z} \circ \mathsf{D}\Delta & \{\textit{adjunction}\} \\
= & k_1 \circ \mathsf{m}_{X,Y} \circ (\mathsf{D}\varepsilon_I \circ \delta_{X,I} \times \mathsf{D}k_2 \circ \delta_{X,Y}) \circ \mathsf{n}_{X,Y \bullet Z} \circ \mathsf{D}\Delta & [\mathrm{C2}] \\
= & k_1 \circ \mathsf{m}_{X,Y} \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}k_2) \circ (\delta_{X,I} \times \delta_{X,Y}) \circ \mathsf{n}_{X,Y \bullet Z} \circ \mathsf{D}\Delta & \{\times \textit{ functoriality}\} \\
= & k_1 \circ \mathsf{m}_{X,Y} \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}k_2) \circ \mathsf{n}_{X,Y} \circ \mathsf{D}\mathsf{n}_{I,Z} \circ \delta_{X \sqcup Y, I \sqcup Z} \circ \mathsf{D}\Delta & \{\textit{colax monoidal } \delta_{X,Y} \text{ (63) } \textit{(p. 130)}\} \\
= & k_1 \circ \mathsf{m}_{X,Y} \circ \mathsf{n}_{X,Y} \circ \mathsf{D}(\varepsilon_I \times k_2) \circ \mathsf{D}\mathsf{n}_{I,Z} \circ \delta_{X \sqcup Y, I \sqcup Z} \circ \mathsf{D}\Delta & \{\mathsf{n} \textit{ naturality}\} \\
= & k_1 \circ \mathsf{D}(\varepsilon_I \times k_2) \circ \mathsf{D}\mathsf{n}_{I,Z} \circ \delta_{X \sqcup Y, I \sqcup Z} \circ \mathsf{D}\Delta & \{\mathsf{m}_{X,Y} \circ \mathsf{n}_{X,Y} = id\} \\
= & k_1 \circ \mathsf{D}((\varepsilon_I \times k_2) \circ \mathsf{n}_{I,Z} \circ \mathsf{D}\Delta) \circ \delta_{X \sqcup Y, I \sqcup Z} & \{\mathsf{D} \textit{ functoriality, } \delta \textit{ naturality}\} \\
= & [\![\Gamma \,?\, (X \sqcup Y) \bullet (I \sqcup Z) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau]\!] &
\end{array}
$$

The coeffect terms here are equal by interchange of $\bullet$ and $\sqcup$:

$$
\begin{array}{lll}
& (X \sqcup Y) \bullet (I \sqcup Z) & \\
= & (X \bullet I) \sqcup (Y \bullet Z) & \{\textit{interchange}\} \\
= & X \sqcup (Y \bullet Z) & \{(C, \bullet, I)\ \textit{monoid}\}
\end{array}
$$

□

Proposition C.4.3 ([*let*-λ-$\equiv$] alternate, restricted)**.**

$$[\![\Gamma \,?\, R \sqcup (R \bullet S) \vdash (\lambda v.e_1)\, e_2 : \tau]\!] \equiv [\![\Gamma \,?\, R \sqcup (R \bullet S) \vdash \textbf{\textit{let}}\ x = e_1\ \textbf{\textit{in}}\ e_2 : \tau]\!]$$

if $\delta_{R,S}$ is colax monoidal (therefore $\bullet$ and $\sqcup$ have interchange), and if $\mathsf{m}_{R,R} = \mathsf{n}_{R,R} = id$ (therefore $\sqcup$ and $\sqcap$ are idempotent).

Proof.

$$
\begin{array}{lll}
& [\![\Gamma \,?\, R \sqcup (R \bullet S) \vdash (\lambda v.e_1)\, e_2 : \tau]\!] & \\
= & \psi(\phi(k_1 \circ \mathsf{m}_{R,R})) \circ (id \times \mathsf{D}k_2 \circ \delta_{R,S}) \circ \mathsf{n}_{R,R \bullet S} \circ \mathsf{D}\Delta & \\
= & k_1 \circ \mathsf{m}_{R,R} \circ (id \times \mathsf{D}k_2 \circ \delta_{R,S}) \circ \mathsf{n}_{R,R \bullet S} \circ \mathsf{D}\Delta & \{\textit{adjunction}\} \\
= & k_1 \circ \mathsf{m}_{R,R} \circ (\mathsf{D}\varepsilon_I \circ \delta_{R,I} \times \mathsf{D}k_2 \circ \delta_{R,S}) \circ \mathsf{n}_{R,R \bullet S} \circ \mathsf{D}\Delta & [\mathrm{C2}] \\
= & k_1 \circ \mathsf{m}_{R,R} \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}k_2) \circ (\delta_{R,I} \times \delta_{R,S}) \circ \mathsf{n}_{R,R \bullet S} \circ \mathsf{D}\Delta & \{\times \textit{ functoriality}\} \\
= & k_1 \circ \mathsf{m}_{R,R} \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}k_2) \circ \mathsf{n}_{R,R} \circ \mathsf{D}\mathsf{n}_{I,S} \circ \delta_{R \sqcup R, I \sqcup S} \circ \mathsf{D}\Delta & \{\textit{colax monoidal } \delta_{R,S} \text{ (63) } \textit{(p. 130)}\} \\
= & k_1 \circ \mathsf{m}_{R,R} \circ \mathsf{n}_{R,R} \circ \mathsf{D}(\varepsilon_I \times k_2) \circ \mathsf{D}\mathsf{n}_{I,S} \circ \delta_{R \sqcup R, I \sqcup S} \circ \mathsf{D}\Delta & \{\mathsf{n} \textit{ naturality}\} \\
= & k_1 \circ \mathsf{D}(\varepsilon_I \times k_2) \circ \mathsf{D}\mathsf{n}_{I,S} \circ \delta_{R, I \sqcup S} \circ \mathsf{D}\Delta & \{\mathsf{m}_{R,R} \circ \mathsf{n}_{R,R} = id\} \\
= & k_1 \circ \mathsf{D}((\varepsilon_I \times k_2) \circ \mathsf{n}_{I,S} \circ \mathsf{D}\Delta) \circ \delta_{R, I \sqcup S} & \{\mathsf{D} \textit{ functoriality, } \delta \textit{ naturality}\} \\
= & [\![\Gamma \,?\, R \bullet (I \sqcup S) \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau]\!] &
\end{array}
$$

For the coeffect terms to be equal, then $\bullet$ and $\sqcup$ must have an interchange law, and $\sqcup$ and $\sqcap$ must both be idempotent such that $\mathsf{m}_{R,R} \circ \mathsf{n}_{R,R} = id$. Note that interchange between $\bullet$ and $\sqcup$ and idempotence of $\sqcup$ implies that $\bullet$ distributes over $\sqcup$ (part of the coeffect algebra definition):

$$
\begin{array}{lll}
& R \bullet (S \sqcup T) & \\
= & (R \sqcup R) \bullet (S \sqcup T) & \{\sqcup \textit{ idempotence}\} \\
= & (R \bullet S) \sqcup (R \bullet T) & \{\textit{interchange}\}
\end{array}
$$

□

β-equivalence.

Lemma C.4.4 ((*let*-β) - Substitution equivalent to **let**)**.**

$$\forall e, e'.\ (\Gamma, x : \tau' \,?\, R \vdash e : \tau) \wedge (\Gamma \,?\, X \vdash e' : \tau') \\ \Rightarrow [\![\Gamma \,?\, R \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau]\!] = [\![\Gamma \,?\, R \bullet (I \sqcup X) \vdash e[x := e'] : \tau]\!]$$

Previously, the substitution lemma was stated with coeffect $R \bullet (R \sqcup X)$ (Lemma 6.3.2, p. 124). Note $R \bullet (I \sqcup X) \equiv R \bullet \sqcup (R \bullet X)$ via distributivity of $\bullet$ over $\sqcup$ (*i.e.*, $\bullet$ is monotone). Here, $R \bullet (I \sqcup X)$ is used as it matches the structure of the categorical semantics, making the proof more clear. Throughout $[\![\Gamma ? X \vdash e' : \tau']\!] = g : \mathsf{D}_X\Gamma \to \tau'$.

▸ [VAR] *i.e.* for $\Gamma, x : \tau' \,?\, I \vdash v : \tau$. There are two cases:

– $x = v$ (therefore $\tau = \tau'$) and the semantics:

$$[\text{VAR}]\ \frac{}{[\![\Gamma, x : \tau' \,?\, I \vdash v : \tau]\!] = \pi_2 \circ \varepsilon_I}$$

By the definition of substitution $x[x := e'] = e'$ therefore proof of (*let*-β) for [VAR] where $v = x$ is then:

$$\begin{array}{lll} & [\![\Gamma \,?\, I \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ x : \tau]\!] & \\ = & \pi_2 \circ \varepsilon_I \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{I,I\sqcup X} & \\ = & \pi_2 \circ (\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \varepsilon_I \circ \delta_{I,I\sqcup X} & \{\varepsilon_I \textit{ naturality}\} \\ = & g \circ \pi_2 \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta & [\text{C2}] \end{array}$$

There are two cases depending on whether the coeffect algebra is top-pointed or bottom-pointed (see syntactic proof in Appendix C.4.2 above).

∗ $\forall R. I \sqsubseteq R$ (lower-bound) then $I \sqcup R = R$ therefore: $X = I \sqcup (I \bullet X)$. Then D is a colax *monoidal* functor, with a counit n_I, which then has the counital property: (see Definition B.1.5)

$$\begin{array}{ccc} 1 \times \mathsf{D}_R A & \xleftarrow{\mathsf{n}_I \times id} & \mathsf{D}_I 1 \times \mathsf{D}_R A \\ {\scriptstyle \pi_2}\downarrow & \swarrow_{\pi_2} & \uparrow{\scriptstyle \mathsf{n}_{I,R}} \\ \mathsf{D}_R A & \xleftarrow[\mathsf{D}\pi_2]{} & \mathsf{D}_{I\sqcup R}(1 \times A) \end{array} \tag{113}$$

Therefore:

$$\begin{array}{lll} = & g \circ \pi_2 \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta & \\ = & g \circ \mathsf{D}\pi_2 \circ \mathsf{D}\Delta & (113) \\ = & g & \{\times \textit{ universal property}\} \end{array}$$

Thus the colax monoidal operation is required to have the counital operation n_I satisfying the counital property of (113).

$*$ $\forall R.R \sqsubseteq I$ (upper-bound) therefore where $I \sqcup X = I$. Therefore:

$$\begin{array}{lll} & g \circ \pi_2 \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta & \\ = & g \circ \pi_2 \circ (id \times \mathsf{D}_{\iota_X^I}) \circ \mathsf{n}_{I,I} \circ \mathsf{D}\Delta & \{I \textit{ is greatest element}\} \\ = & g \circ \mathsf{D}_{\iota_X^I} \circ \pi_2 \circ \mathsf{n}_{I,I} \circ \mathsf{D}\Delta & \{\pi_2 \textit{ naturality}\} \\ = & g \circ \mathsf{D}_{\iota_X^I} \circ \pi_2 \circ \Delta & \{n_{R,R} \textit{ idempotence}\} \\ = & g \circ \mathsf{D}_{\iota_X^I} & \{\times \textit{ universal property}\} \end{array}$$

where $\mathsf{D}_{\iota_X^I} : \mathsf{D}_I \stackrel{\cdot}{\to} \mathsf{D}_X$.

– $x \neq v$: thus the semantics is as follows where $|\Gamma| = n$ and v is the i^{th} variable in Γ:

$$[\textsc{var}]\frac{}{[\![\Gamma, x : \tau' \,?\, I \vdash v : \tau]\!] = \pi_2 \circ \pi_1^{(n-i+1)} \circ \varepsilon_I}$$

By the definition of substitution $v[x := e'] = v$ therefore proof of (*let-*β) for [VAR] where $v \neq x$ is then when:

$$[\![\Gamma \,?\, (I \bullet (I \sqcup X)) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ v : \tau]\!] \equiv [\![\Gamma \,?\, (I \bullet (I \sqcup X)) \vdash v : \tau]\!]$$

There are two cases depending on whether the coeffect algebra is top-pointed or bottom-pointed (see syntactic proof in Appendix C.4.2 above).

$*$ $\forall R.I \sqsubseteq R$ (lower-bound) then $I \sqcup R = R$ therefore: $X = I \sqcup (I \bullet X)$ and therefore [SUB] is used in the type/coeffect derivation for v:

$$\begin{aligned} [\![\Gamma ? (I \bullet (I \sqcup X)) \vdash v : \tau]\!] &\equiv [\![\Gamma ? I \vdash v : \tau]\!] \circ \mathsf{D}_{\iota_I^X} \\ &\equiv \pi_2 \circ \pi_1^{n-i} \circ \varepsilon_I \circ \mathsf{D}_{\iota_I^X} \end{aligned}$$

The proof is then:

$$\begin{array}{lll} & [\![\Gamma \,?\, (I \bullet (I \sqcup X)) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ v : \tau]\!] & \\ = & \pi_2 \circ \pi_1^{(n-i+1)} \circ \varepsilon_I \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{I,I\sqcup X} & \\ = & \pi_2 \circ \pi_1^{(n-i+1)} \circ (\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \varepsilon_I \circ \delta_{I,I\sqcup X} & \{\varepsilon_I \textit{ naturality}\} \\ = & \pi_2 \circ \pi_1^{(n-i+1)} \circ (\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta & [\text{C2}] \\ = & \pi_2 \circ \pi_1^{(n-i+1)} \circ (\varepsilon_I \times g) \circ \mathsf{D}_{\iota_I^X} \circ \mathsf{n}_{X,X} \circ \mathsf{D}\Delta & \{I \textit{ is least element}\} \\ = & \pi_2 \circ \pi_1^{(n-i+1)} \circ (\varepsilon_I \times g) \circ \mathsf{D}_{\iota_I^X} \circ \Delta & \{\mathsf{n}_{X,X} \textit{ idempotence}\} \\ = & \pi_2 \circ \pi_1^{(n-i)} \circ \pi_1 \circ (\varepsilon_I \times g) \circ \mathsf{D}_{\iota_I^X} \circ \Delta & \{\textit{composition}\} \\ = & \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon_I \circ \pi_1 \circ \mathsf{D}_{\iota_I^X} \circ \Delta & \{\times \textit{ universality}\} \\ = & \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon_I \circ \mathsf{D}_{\iota_I^X} \circ \pi_1 \circ \Delta & \{\mathsf{D}_\iota \textit{ naturality}\} \\ = & \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon_I \circ \mathsf{D}_{\iota_I^X} & \{\times \textit{ universality}\} \\ = & [\![\Gamma ? (I \bullet (I \sqcup X)) \vdash v : \tau]\!] \quad \square & \end{array}$$

$*$ $\forall R.R \sqsubseteq I$ (upper-bound) therefore where $I \sqcup X = I$. Therefore:

$$\begin{aligned} [\![\Gamma ? (I \bullet (I \sqcup X)) \vdash v : \tau]\!] &\equiv [\![\Gamma ? I \vdash v : \tau]\!] \\ &\equiv \pi_2 \circ \pi_1^{n-i} \circ \varepsilon_I \end{aligned}$$

$$
\begin{aligned}
& [\![\Gamma \,?\, (I \bullet (I \sqcup X)) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ x : \tau]\!] \\
= \;& \pi_2 \circ \pi_1^{(n-i+1)} \circ \varepsilon_I \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{I,I\sqcup X} \\
= \;& \pi_2 \circ \pi_1^{(n-i+1)} \circ (\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \varepsilon_I \circ \delta_{I,I\sqcup X} && \{\varepsilon_I \ \textit{naturality}\} \\
= \;& \pi_2 \circ \pi_1^{(n-i+1)} \circ (\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta && [\mathrm{C2}] \\
= \;& \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon_I \circ \pi_1 \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta && \{\times \ \textit{universality}\} \\
= \;& \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon_I \circ \pi_1 \circ \mathsf{n}_{I,I} \circ \mathsf{D}\Delta && \{I \sqcup X = I\} \\
= \;& \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon_I \circ \pi_1 \circ \Delta && \{\mathsf{n}_{I,I} \ \textit{idempotence}\} \\
= \;& \pi_2 \circ \pi_1^{(n-i)} \circ \varepsilon_I && \{\times \ \textit{universality}\} \\
= \;& [\![\Gamma ? I \vdash v : \tau]\!]
\end{aligned}
\tag{114}
$$

▸ [ABS] – As discussed in Section 6.3.1 (p. 121), β-equality requires an interchange law between $\bullet$ and $\sqcap$, which follows from the proof in Appendix C.4.2 (p. 204). The proof here then assumes this property, which is also required such that $\delta_{R,S}$ is a colax monoidal indexed comultiplication (used in the proof) (see Definition 6.4.3, p. 131).

$$
[\textsc{abs}]\frac{[\![\Gamma, x : \tau', v : \sigma \,?\, R \sqcap S \vdash e : \tau]\!] = f : \mathsf{D}_{R\sqcap S}((\Gamma \times \tau') \times \sigma) \to \tau}{[\![\Gamma, x : \tau' \,?\, R \vdash \lambda v.e : \sigma \xrightarrow{S} \tau]\!] = \phi(f \circ \mathsf{m}) : \mathsf{D}_R(\Gamma \times \tau') \to (\mathsf{D}_S\sigma \Rightarrow \tau)}
$$

By the definition of substitution, $(\lambda v.e)[x := e'] = \lambda v.e[x := e']$, the inductive hypothesis is therefore:

$$
(A) = [\textsc{weak}]\frac{[\![\Gamma \,?\, X \vdash e' : \tau']\!] = g : \mathsf{D}_X\Gamma \to \tau'}{[\![\Gamma, v : \sigma \,?\, X \vdash e' : \tau']\!] = g \circ \mathsf{D}\pi_1 : \mathsf{D}_X(\Gamma \times \sigma) \to \tau'} \tag{115}
$$

$$
\frac{[\textsc{exch}]\ \dfrac{[\![\Gamma, x : \tau', v : \sigma \,?\, R \sqcap S \vdash e : \tau]\!] = f : \mathsf{D}_{R\sqcap S}((\Gamma \times \tau') \times \sigma) \to \tau}{[\![\Gamma, v : \sigma, x : \tau' \,?\, R \sqcap S \vdash e : \tau]\!] = f \circ \mathsf{D}\chi : \mathsf{D}_{R\sqcap S}((\Gamma \times \sigma) \times \tau') \to \tau} \qquad (A)}{[\![\Gamma, v : \sigma \,?\, (R \sqcap S) \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e : \tau]\!]} \tag{116}
$$

$$
\begin{aligned}
= \;& (f \circ \mathsf{D}\chi) \circ D((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \circ n_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R\sqcap S, I\sqcup X} : \mathsf{D}_{(R\sqcap S)\bullet(I\sqcup X)}(\Gamma \times \sigma) \to \tau \\
= \;& [\![\Gamma, v : \sigma \,?\, (R \sqcap S) \bullet (I \sqcup X) \vdash e[x := e'] : \tau]\!] = h : \mathsf{D}_{(R\sqcap S)\bullet(I\sqcup X)}(\Gamma \times \sigma) \to \tau
\end{aligned}
$$

i.e.. $h = f \circ \mathsf{D}\chi \circ D((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \circ n_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R\sqcap S, I\sqcup X}$.

The conclusion of (*let-*β) for [ABS] is thus:

$$
\begin{aligned}
& [\![\Gamma \,?\, R \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ \lambda v.e : \sigma \xrightarrow{S} \tau]\!] \\
= \;& \phi(f \circ \mathsf{m}_{R,S}) \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R,I\sqcup X} : \mathsf{D}_{R\bullet(I\sqcup X)}\Gamma \to (\mathsf{D}_S\sigma \Rightarrow \tau) && (117) \\
= \;& [\textsc{abs}]\frac{[\![\Gamma, v : \sigma \,?\, P \sqcap Q \vdash e[x := e'] : \tau]\!] = h : \mathsf{D}_{P\sqcap Q}(\Gamma \times \sigma) \to \tau}{[\![\Gamma \,?\, P \vdash \lambda v.(e[x := e']) : \sigma \xrightarrow{Q} \tau]\!] = \phi(h \circ \mathsf{m}_{P,Q}) : \mathsf{D}_P\Gamma \to (\mathsf{D}_Q\sigma \Rightarrow \tau)} && (118)
\end{aligned}
$$

where $P \sqcap Q = (R \sqcap S) \bullet (I \sqcup X)$, $P = R \bullet (I \sqcup X)$ and $Q = S$ therefore $(R \sqcap S) \bullet (I \sqcup X) = (R \bullet (I \sqcup X)) \sqcap S$. The earlier inductive proof for the substitution lemma (p. 204) showed that this holds when $\bullet$ has an interchange law with $\sqcap$ and $(C, I, \sqcap)$ is a monoid.

The semantics uses the χ natural transformation for exchange where $\chi = \langle \pi_1 \times id, \pi_2 \circ \pi_1 \rangle : (A \times B) \times C \to (A \times C) \times B$, with naturality $\chi \circ (f \times g) \times h = (f \times h) \times g \circ \chi$.

$$
\begin{aligned}
&[\![\Gamma \,?\, P \vdash \lambda v.e[x := e'] : \sigma \xrightarrow{Q} \tau]\!] \\
&= \phi(h \circ \mathsf{m}_{P,Q}) \\
&= \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcap S, I \sqcup X} \circ \mathsf{m}_{R \sqcap S, I \sqcup X} \\
&= \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \mathsf{D}\mathsf{m}_{I \sqcup X, I} \circ \mathsf{m}_{R,S} \circ (\delta_{R, I \sqcup S} \times \delta_{S,I}) && \{\textit{lax monoidal } \delta_{R,S}\} \\
&= \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}(((\varepsilon_I \times \varepsilon_I) \circ \mathsf{n}_{I,I}) \times (g \circ \mathsf{D}\pi_1)) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \mathsf{D}\mathsf{m}_{I \sqcup X, I} \circ \ldots && \{\textit{colax monoidal } \varepsilon_I\} \\
&= \phi(f \circ \mathsf{D}\chi \circ \mathsf{D}(((\varepsilon_I \times \varepsilon_I) \times (g \circ \mathsf{D}\pi_1))) \circ \mathsf{D}(\mathsf{n}_{I,I} \times \mathit{id}) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \mathsf{m}_{I \sqcup X, I}) \circ \ldots && \{\times \textit{ functoriality}\} \\
&= \phi(f \circ \mathsf{D}(((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \times \varepsilon_I)) \circ \chi \circ (\mathsf{n}_{I,I} \times \mathit{id}) \circ \ldots && \{\chi \textit{ naturality}\} \\
&= \phi(f \circ \mathsf{D}(((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \times \varepsilon_I) \circ ((\pi_1 \times \mathit{id}) \times (\pi_1 \circ \pi_1)) \circ \Delta \circ (\mathsf{n}_{I,I} \times \mathit{id}) \ldots && \{\chi \textit{ def}\} \\
&= \phi(f \circ \mathsf{D}(((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \times \varepsilon_I) \circ (((\pi_1 \circ \mathsf{n}_{I,I}) \times \mathit{id}) \times (\pi_2 \circ \pi_1 \circ (\mathsf{n}_{I,I} \times \mathit{id}))) \circ \Delta) \ldots && \{\Delta \textit{ naturality}\} \\
&= \phi(f \circ \mathsf{D}(((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \times \varepsilon_I) \circ (((\pi_1 \circ \mathsf{n}_{I,I}) \times \mathit{id}) \times (\pi_2 \circ \mathsf{n}_{I,I} \circ \pi_1)) \circ \Delta) \ldots && \{\pi_1 \textit{ naturality}\} \\
&= \phi(f \circ \mathsf{D}(((\varepsilon_I \times (g \circ \mathsf{D}\pi_1)) \times \varepsilon_I) \circ (((\mathsf{D}\pi_1) \times \mathit{id}) \times (\mathsf{D}\pi_2 \circ \pi_1)) \circ \Delta) \ldots && \{(113) \textit{ counitality}\} \\
&= \phi(f \circ \mathsf{D}((((\varepsilon_I \circ \mathsf{D}\pi_1) \times (g \circ \mathsf{D}\pi_1)) \times (\varepsilon_I \circ \mathsf{D}\pi_2 \circ \pi_1)) \circ \Delta \circ (n_{I,X} \circ \mathsf{D}\Delta \circ \mathsf{m}_{I \sqcup X, I})) \ldots && \{\times \textit{ functoriality}\} \\
&= (\mathit{left}) \ldots ((\varepsilon_I \circ \mathsf{D}\pi_1) \times (g \circ \mathsf{D}\pi_1)) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \mathsf{m}_{I \sqcup X, I} \ldots && \{\Delta \textit{ naturality}\} \\
&= \ldots ((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}(\pi_1 \times \pi_1) \circ \mathsf{D}\Delta \circ \mathsf{m}_{I \sqcup X, I} \ldots && \{\mathsf{n}_{R,S} \textit{ naturality}\} \\
&= \ldots ((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \mathsf{D}\pi_1 \circ \mathsf{m}_{I \sqcup X, I} \ldots && \{\Delta \textit{ naturality}\} \\
&= \ldots ((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \pi_1 \ldots && \{\textit{dual of } (113), \textit{ unitality}\} \\
&= (\mathit{right}) \ldots \varepsilon_I \circ \mathsf{D}\pi_2 \circ \pi_1 \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta \circ \mathsf{m}_{I \sqcup X, I} \ldots \\
&= \ldots \varepsilon_I \circ \pi_2 \ldots \\
&= \phi(f \circ \mathsf{D}((((\varepsilon_I \times g) \circ \mathsf{n}_{I,x} \circ \mathsf{D}\Delta\pi_1) \times (\varepsilon_I \circ \pi_2)) \circ \Delta) \circ \mathsf{m}_{R,S} \circ (\delta_{R, I \sqcup S} \times \delta_{S,I})) && \{\textit{(left) and (right)}\} \\
&= \phi(f \circ \mathsf{D}(((\varepsilon_I \times g) \circ \mathsf{n}_{I,x} \circ \mathsf{D}\Delta) \times \varepsilon_I) \circ \mathsf{m}_{R,S} \circ (\delta_{R, I \sqcup S} \times \delta_{S,I})) && \{\times \textit{ universality}\} \\
&= \phi(f \circ \mathsf{m}_{R,S} \circ (\mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,x} \circ \mathsf{D}\Delta) \times \mathsf{D}\varepsilon_I) \circ (\delta_{R, I \sqcup S} \times \delta_{S,I})) && \{\mathsf{m}_{R,S} \textit{ naturality}\} \\
&= \phi(f \circ \mathsf{m}_{R,S} \circ (\mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,x} \circ \mathsf{D}\Delta) \circ \delta_{R, I \sqcup S}) \times \mathit{id}) && \{\times \textit{ universality, [C2]}\} \\
&= \phi(f \circ \mathsf{m}_{R,S}) \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,x} \circ \mathsf{D}\Delta) \circ \delta_{R, I \sqcup S} && \{\textit{adjunction}\} \\
&= [\![\Gamma \,?\, R \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e' \ \mathbf{in}\ \lambda v.e : \sigma \xrightarrow{S} \tau]\!]
\end{aligned}
$$

▸ [APP] Case proving $[\![\Gamma ? (R \sqcup (S \bullet T)) \bullet (I \sqcup S) \vdash \mathbf{let}\, x = e' \,\mathbf{in}\, e_1 e_2 \equiv (e_1\, e_2)[x := e']]\!]$, where $e_1\, e_2$ has derivation:

$$
[\text{APP}]\frac{\begin{array}{c}[\![\Gamma, x : \tau' \,?\, X \vdash e_1 : \sigma \xrightarrow{S} \tau]\!] = k_1 : \mathsf{D}_R(\Gamma \times \tau') \to (\mathsf{D}_S\sigma \Rightarrow \tau) \\ [\![\Gamma, x : \tau' \,?\, T \vdash e_2 : \sigma]\!] = k_2 : \mathsf{D}_Z\Gamma \to \sigma\end{array}}{\begin{array}{c}[\![\Gamma, x : \tau' \,?\, R \sqcup (S \bullet T) \vdash e_1\, e_2 : \tau]\!] \\ = \psi k_1 \circ (\mathit{id} \times \mathsf{D}k_2 \circ \delta_{S,T}) \circ \mathsf{n}_{R, S \bullet T} \circ \mathsf{D}\Delta : \mathsf{D}_{R \sqcup (S \bullet T)}(\Gamma \times \tau') \to \tau\end{array}}
$$

By the definition of substitution $(e_1\, e_2)[x := e'] = e_1[x := e']\, e_2[x := e']$. The inductive hypotheses are therefore:

$$
\begin{aligned}
&[\![\Gamma ? R \bullet (I \sqcup X) \vdash e_1[x := e'] : \sigma \xrightarrow{S} \tau]\!] = h_1 : \mathsf{D}_{R \bullet (I \sqcup X)}\Gamma \to (\mathsf{D}_S\sigma \Rightarrow \tau) \\
\equiv\ &[\![\Gamma ? R \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e' \ \mathbf{in}\ e_1 : \sigma \xrightarrow{S} \tau]\!] = k_1 \circ \mathsf{D}((\varepsilon_I \circ g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R, I \sqcup X} \qquad (119)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\Gamma ? T \bullet (I \sqcup X) \vdash e_2[x := e'] : \sigma]\!] = h_2 : \mathsf{D}_{T \bullet (I \sqcup X)}\Gamma \to \sigma \\
\equiv\ &[\![\Gamma ? T \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e' \ \mathbf{in}\ e_2 : \sigma]\!] = k_2 \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{T, I \sqcup X} \qquad (120)
\end{aligned}
$$

The conclusion of (*let-β*) for [APP] is thus:

$$\begin{aligned}
&[\![\Gamma \vdash e_1[x := e']\, e_2[x := e'] : \tau]\!] \\
= \;& \psi h_1 \circ (id \times (\mathsf{D}h_2 \circ \delta_{S,T\bullet(I\sqcup X)})) \circ \mathsf{n}_{R\bullet(I\sqcup X),S\bullet(T\bullet(I\sqcup X))} \circ \mathsf{D}\Delta \\
= \;& [\![\Gamma \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e_1\, e_2 : \tau]\!] \\
= \;& \psi k_1 \circ (id \times \mathsf{D}k_2 \circ \delta_{S,T}) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\Delta \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R\sqcup(S\bullet T),I\sqcup X} \qquad (121)
\end{aligned}$$

with the following proof:

$$\begin{aligned}
&[\![\Gamma \vdash (e_1\, e_2)[x := e'] : \tau]\!] \\
=\;& \psi h_1 \circ (id \times (\mathsf{D}h_2 \circ \delta_{S,T\bullet(I\sqcup X)})) \circ \mathsf{n}_{R\bullet(I\sqcup X),S\bullet(T\bullet(I\sqcup X))} \circ \mathsf{D}\Delta && (121)\\
=\;& \psi h_1 \circ (id \times (\mathsf{D}(k_2 \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{T,I\sqcup X}) \circ \delta_{S,T\bullet(I\sqcup X)})) \circ \mathsf{n} \ldots && (120)\\
=\;& \psi h_1 \circ (id \times (\mathsf{D}(k_2 \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta)) \circ \mathsf{D}\delta_{T,I\sqcup X} \circ \delta_{S,T\bullet(I\sqcup X)})) \circ \mathsf{n} \ldots && \{\mathsf{D}\ \textit{functor}\}\\
=\;& \psi h_1 \circ (id \times (\mathsf{D}(k_2 \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta)) \circ \delta_{S,T} \circ \delta_{S\bullet T,I\sqcup X})) \circ \mathsf{n} \ldots && [\mathrm{C3}]\\
=\;& \psi(k_1 \circ \mathsf{D}((\varepsilon_I \circ g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R,I\sqcup X}) \circ (id \times (\ldots \circ \delta_{S,T} \circ \delta_{S\bullet T,I\sqcup X})) \circ \mathsf{n} \ldots && (119)\\
=\;& \psi k_1 \circ ((\mathsf{D}((\varepsilon_I \circ g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R,I\sqcup X}) \times (\ldots \circ \delta_{S,T} \circ \delta_{S\bullet T,I\sqcup X})) \circ \mathsf{n} \ldots && \{\textit{adjunction}\}\\
=\;& \psi k_1 \circ (\mathsf{D}(\ldots \circ \mathsf{D}\Delta) \times (\ldots \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\mathsf{n}_{I\sqcup X,I\sqcup X} \circ \delta_{R\sqcup(S\bullet T),(I\sqcup X)\sqcup(I\sqcup X)} \circ \ldots && \{\delta\ \textit{colax monoidal}\}\\
=\;& \psi k_1 \circ (\mathsf{D}(\ldots \circ \mathsf{D}\Delta) \times (\ldots \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\mathsf{n}_{I\sqcup X,I\sqcup X} \circ \mathsf{D}\mathsf{D}\Delta \circ \delta_{\ldots} && \{\delta\ \textit{naturality}\}\\
=\;& \psi k_1 \circ (\mathsf{D}(\ldots \circ \mathsf{D}\Delta) \times (\ldots \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}(\mathsf{n}_{I\sqcup X,I\sqcup X} \circ \Delta) \circ \delta_{R\sqcup(S\bullet T),(I\sqcup X)\sqcup(I\sqcup X)} && \{\mathsf{D}\ \textit{functor}\}\\
=\;& \psi k_1 \circ (\mathsf{D}(\ldots \circ \mathsf{D}\Delta) \times (\ldots \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\Delta \circ \delta_{R\sqcup(S\bullet T),(I\sqcup X)} && \{\mathsf{n}\ \textit{idempotent}\}\\
=\;& \psi k_1 \circ (\mathsf{D}(\ldots \circ \mathsf{D}\Delta) \times (\mathsf{D}(k_2 \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta)) \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\Delta \circ \delta_{\ldots} &&\\
=\;& \psi k_1 \circ (\mathsf{D}(\ldots \circ \mathsf{D}\Delta) \times (\mathsf{D}k_2 \circ \mathsf{D}\mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\Delta \circ \delta_{\ldots} && \{\mathsf{D}\ \textit{functor}\}\\
=\;& \psi k_1 \circ (\mathsf{D}(\ldots \circ \mathsf{D}\Delta) \times (\mathsf{D}k_2 \circ \delta_{S,T} \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta))) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\Delta \circ \delta_{\ldots} && \{\delta\ \textit{naturality}\}\\
=\;& \psi k_1 \circ (\mathsf{D}(\varepsilon_I \circ g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \times (\mathsf{D}k_2 \circ \delta_{S,T} \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta))) \circ \mathsf{n}_{R,S\bullet T} \circ \ldots &&\\
=\;& \psi k_1 \circ (id \times (\mathsf{D}k_2 \circ \delta_{S,T})) \circ (\hat{\Delta}(\mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta))) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\Delta \circ \delta_{\ldots} && \{\hat{\Delta}A = A \times A\}\\
=\;& \psi k_1 \circ (id \times (\mathsf{D}k_2 \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}(\hat{\Delta}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta)) \circ \mathsf{D}\Delta \circ \delta_{R\sqcup(S\bullet T),(I\sqcup X)} && \{\mathsf{n}\ \textit{naturality}\}\\
=\;& \psi k_1 \circ (id \times (\mathsf{D}k_2 \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}(\hat{\Delta}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \Delta) \circ \delta_{R\sqcup(S\bullet T),(I\sqcup X)} && \{\mathsf{D}\ \textit{functor}\}\\
=\;& \psi k_1 \circ (id \times (\mathsf{D}k_2 \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}(\Delta \circ (\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R\sqcup(S\bullet T),(I\sqcup X)} && \{\Delta\ \textit{naturality}\}\\
=\;& \psi k_1 \circ (id \times (\mathsf{D}k_2 \circ \delta_{S,T})) \circ \mathsf{n}_{R,S\bullet T} \circ \mathsf{D}\Delta \circ \mathsf{D}((\varepsilon_I \times g) \circ \mathsf{n}_{I,X} \circ \mathsf{D}\Delta) \circ \delta_{R\sqcup(S\bullet T),(I\sqcup X)} && \{\Delta\ \textit{naturality}\}\\
=\;& [\![\Gamma?(R \sqcup (S \bullet T)) \bullet (I \sqcup X) \vdash \mathbf{let}\ x = e'\ \mathbf{in}\ e_1\, e_2 : \tau]\!] \quad \square && (121)
\end{aligned}$$

Proposition C.4.5 (equality of refined and derived semantics for [APP], see p. 131)**.** *The* refined *semantics for application in the coeffect calculus (Section 6.4.2, p. 129) is equal to the derived semantics for application when $\delta_{R,S}$ is a colax monoidal operation (Definition 6.4.3, equation 63) and thus when $\bullet$ and $\sqcup$ have an interchange law ($(R \bullet X) \sqcup (S \bullet Y) = (R \sqcup S) \bullet (X \sqcup Y)$).*

Proof.

$$
\begin{aligned}
& [\![\Gamma \,?\, R \sqcup (S \bullet T) \vdash e_1\, e_2 : \tau]\!]^{refined} \\
= \;& \psi k_1 \circ (id \times k_2^{\dagger S,T}) \circ \mathsf{n}_{R,S \bullet T} \circ \mathsf{D}\Delta \\
= \;& \psi k_1 \circ (id \times (\mathsf{D}k_2 \circ \delta_{S,T})) \circ \mathsf{n}_{R,S \bullet T} \circ \mathsf{D}\Delta && \{\mathsf{D}f \circ \delta_{X,Y} = f^{\dagger X,Y}\} \\
= \;& \psi k_1 \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}k_2) \circ (\delta_{R,I} \times \delta_{S,T}) \circ \mathsf{n}_{R \bullet I, S \bullet T} \circ \mathsf{D}\Delta && [\mathrm{C2}] \\
= \;& \psi k_1 \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}k_2) \circ \mathsf{n}_{R,S} \circ \mathsf{D}\mathsf{n}_{I,T} \circ \delta_{R \sqcup S, I \sqcup T} \circ \mathsf{D}\Delta && \{\textit{colax monoidal } \delta \text{ (63)}\} \\
= \;& \psi k_1 \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}k_2) \circ \mathsf{n}_{R,S} \circ \mathsf{D}\mathsf{n}_{I,T} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcup S, I \sqcup T} && \{\delta_{R,S} \textit{ naturality}\} \\
= \;& \psi k_1 \circ \mathsf{n}_{R,S} \circ \mathsf{D}((\varepsilon_I \times k_2) \circ \mathsf{n}_{I,T} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcup S, I \sqcup T} && \{\mathsf{n}_{R,S} \textit{ naturality}\} \\
= \;& \psi k_1 \circ \mathsf{n}_{R,S} \circ ((\varepsilon_I \times k_2) \circ \mathsf{n}_{I,T} \circ \mathsf{D}\Delta)^{\dagger R \sqcup S, I \sqcup T} && \{\mathsf{D}f \circ \delta_{X,Y} = f^{\dagger X,Y}\} \\
= \;& [\![\Gamma \,?\, (R \sqcup S) \bullet (I \sqcup T) \vdash e_1\, e_2 : \tau]\!]^{derived}
\end{aligned}
$$

The two coeffect terms are equal by interchange $(R \sqcup S) \bullet (I \sqcup T) \equiv (R \bullet I) \sqcup (S \bullet T)$. □

Proposition C.4.6. *In the coeffect calculus, η-equivalence holds for the refined semantics (Section 6.4, p. 126) given a colax monoidal indexed comonad and when $\mathsf{n}_{R,S} \circ \mathsf{m}_{R,S} = id$ with the requirement that $\sqcup = \sqcap$, i.e.,*

$$\mathsf{D}_R A \times \mathsf{D}_S B \xrightarrow{\mathsf{m}^{R,S}_{A,B}} \mathsf{D}_{R \sqcap S}(A \times B) \equiv \mathsf{D}_{R \sqcup S}(A \times B) \xrightarrow{\mathsf{n}^{R,S}_{A,B}} \mathsf{D}_R A \times \mathsf{D}_S B$$

Proof. For $\Gamma ? R \vdash f \equiv (\lambda x.fx) : \sigma \xrightarrow{S} \tau$.

$$
\begin{aligned}
& [\![\Gamma \,?\, R \vdash (\lambda x.fx) : \sigma \xrightarrow{S} \tau]\!] \\
= \;& \phi(\psi([\![f]\!] \circ \mathsf{D}\pi_1) \circ (id \times (\pi_2 \circ \varepsilon_I)^{\dagger S,I}) \circ \mathsf{n}_{R,S} \circ \mathsf{D}\Delta \circ \mathsf{m}_{R,S}) \\
= \;& \phi(\psi([\![f]\!] \circ \mathsf{D}\pi_1) \circ (id \times (\mathsf{D}(\pi_2 \circ \varepsilon_I) \circ \delta_{S,I})) \circ \mathsf{n}_{R,S \bullet I} \circ \mathsf{D}\Delta \circ \mathsf{m}_{R,S}) \\
= \;& \phi(\psi([\![f]\!] \circ \mathsf{D}\pi_1) \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}(\pi_2 \circ \varepsilon_I)) \circ (\delta_{R,I} \times \delta_{S,I}) \circ \mathsf{n}_{R \bullet I, S \bullet I} \circ \mathsf{D}\Delta \circ \mathsf{m}_{R,S}) && [\mathrm{C2}] \\
= \;& \phi(\psi([\![f]\!] \circ \mathsf{D}\pi_1) \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}(\pi_2 \circ \varepsilon_I)) \circ \mathsf{n}_{R,S} \circ \mathsf{D}\mathsf{n}_{I,I} \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{D}\Delta \circ \mathsf{m}_{R,S}) && \{\textit{colax monoidal } \delta\} \\
= \;& \phi(\psi([\![f]\!] \circ \mathsf{D}\pi_1) \circ (\mathsf{D}\varepsilon_I \times \mathsf{D}(\pi_2 \circ \varepsilon_I)) \circ \mathsf{n}_{R,S} \circ \mathsf{D}\mathsf{n}_{I,I} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\delta_{R,S} \textit{ naturality}\} \\
= \;& \phi(\psi([\![f]\!] \circ \mathsf{D}\pi_1) \circ \mathsf{n}_{R,S} \circ \mathsf{D}((\varepsilon_I \times (\pi_2 \circ \varepsilon_I)) \circ \mathsf{n}_{I,I} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\mathsf{n}_{R,S} \textit{ naturality}\} \\
= \;& \phi(\psi([\![f]\!]) \circ (\mathsf{D}\pi_1 \times id) \circ \mathsf{n}_{R,S} \circ \mathsf{D}((\varepsilon_I \times (\pi_2 \circ \varepsilon_I)) \circ \mathsf{n}_{I,I} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\textit{adjunction}\} \\
= \;& \phi(\psi([\![f]\!]) \circ \mathsf{n}_{R,S} \circ \mathsf{D}(\pi_1 \times id) \circ \mathsf{D}((\varepsilon_I \times (\pi_2 \circ \varepsilon_I)) \circ \mathsf{n}_{I,I} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\mathsf{n}_{R,S} \textit{ naturality}\} \\
= \;& \phi(\psi([\![f]\!]) \circ \mathsf{n}_{R,S} \circ \mathsf{D}((\pi_1 \times \pi_2) \circ (\varepsilon_I \times \varepsilon_I) \circ \mathsf{n}_{I,I} \circ \mathsf{D}\Delta) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\textit{functor}\} \\
= \;& \phi(\psi([\![f]\!]) \circ \mathsf{n}_{R,S} \circ \mathsf{D}((\pi_1 \times \pi_2) \circ (\varepsilon_I \times \varepsilon_I) \circ \Delta) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\mathsf{n}_{I,I} \textit{ idempotence}\} \\
= \;& \phi(\psi([\![f]\!]) \circ \mathsf{n}_{R,S} \circ \mathsf{D}((\pi_1 \times \pi_2) \circ \Delta \circ \varepsilon_I) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\Delta \textit{ naturality}\} \\
= \;& \phi(\psi([\![f]\!]) \circ \mathsf{n}_{R,S} \circ \mathsf{D}(\varepsilon_I) \circ \delta_{R \sqcup S, I \sqcup I} \circ \mathsf{m}_{R,S}) && \{\times \textit{ universality}\} \\
= \;& \phi(\psi([\![f]\!]) \circ \mathsf{n}_{R,S} \circ \mathsf{m}_{R,S}) && [\mathrm{C2}] \\
= \;& \phi(\psi([\![f]\!])) && \{\mathsf{n}_{R,S} \circ \mathsf{m}_{R,S} = id\} \\
= \;& [\![f]\!] && \{\textit{adjunction}\} \\
= \;& [\![\Gamma ? R \vdash f : \sigma \xrightarrow{S} \tau]\!]
\end{aligned}
$$

□

Appendix D

Haskell and GHC type-system features

This appendix summarising Haskell/GHC type-system features is based on a similar appendix appearing in the author's paper "*Efficient and Correct Stencil Computation via Pattern Matching and Static Typing*" appearing at *DSL 2011 (IFIP Working Conference on Domain-Specific Languages)* [**OM11**]. Section D.4 is based on "*Mathematical Structures for Data Types with Restricted Parametricity*" which appeared in pre-proceedings of *TFP 2012 (Trends in Functional Programming)* [**OM12a**].

D.1. Type classes

Type classes in Haskell provide a mechanism for *ad-hoc polymorphism*, also known as overloading or *type-indexed functions* [**HHPJW96**]. Consider the equality operation ($\equiv$). Many types have a well-defined notion of equality, thus an overloaded ($\equiv$) operator is useful, where the argument type determines the actual equality operation used. The equality type class is defined:

$$\textbf{class } \mathit{Eq}\ a \textbf{ where}$$
$$(\equiv) :: a \to a \to \mathit{Bool}$$

A type is declared a member of a class by an *instance* definition, which provides definitions of the class functions for a particular type. For example:

$$\textbf{instance } \mathit{Eq}\ \mathit{Int} \textbf{ where}$$
$$x \equiv y = \mathit{eqInt}\ x\ y$$

where *eqInt* is the built-in equality operation for *Int*. By this instance definition, *Int* is a member of *Eq*. Usage of ($\equiv$) on polymorphically-typed arguments imposes a *type-class constraint* on the polymorphic type of the arguments. Type-class constraints enforce that a type is an instance of a class and are written on the left-hand side of a type, preceding an arrow $\Rightarrow$. For example, the following function:

$$f\ x\ y = x \equiv y$$

has type $f :: \mathit{Eq}\ a \Rightarrow a \to a \to \mathit{Bool}$, where $\mathit{Eq}\ a$ is the type-class constraint that a be a member of *Eq*.

D.2. GADTs

Generalised algebraic data types, or GADTs, [**PJWW04**, **PJVWW06**] have become a powerful technique for dependent-type like programming in Haskell. Consider the following algebraic data type in Haskell which encodes a simple term calculus of products and projections over some type:

$$\textbf{data } \mathit{Term}\ a = \mathit{Pair}\ (\mathit{Term}\ a)\ (\mathit{Term}\ a) \mid \mathit{Fst}\ (\mathit{Term}\ a) \mid \mathit{Snd}\ (\mathit{Term}\ a) \mid \mathit{Val}\ a$$

This is polymorphic in the type a and recursive. For each *tag* in a the definition of an ADT (*e.g.*, *Pair*, *Fst*, etc.) a constructor is generated; *e.g.*, for the first two data constructors of *Term*:

$$Pair :: Term\ a \to Term\ a \to Term\ a$$
$$Fst :: Term\ a \to Term\ a$$

Thus, every constructor returns a value of type $Term\ a$. We can define another ADT of values *Val* and write an evaluator from *Term* to *Val*:

$$\textbf{data}\ Val\ a = ValPair\ (Val\ a)\ (Val\ a) \mid ValConst\ a$$
$$eval :: Term\ a \to Val\ a$$
$$eval\ (Pair\ x\ y) = ValPair\ (eval\ x)\ (eval\ y)$$
$$eval\ (Fst\ x) \quad = \textbf{case}\ (eval\ x)\ \textbf{of}\ ValPair\ a\ b \to a; \quad \_ \to error\ \texttt{"whoops"}$$
$$eval\ (Snd\ x) \quad = \textbf{case}\ (eval\ x)\ \textbf{of}\ ValPair\ a\ b \to b; \quad \_ \to error\ \texttt{"whoops"}$$
$$eval\ (Val\ x) \quad = ValConst\ x$$

Unfortunately, the *Term* ADT allows non-sensical terms in the *Term* calculus to be constructed, such as $Fst\ (Val\ 1)$, therefore error handling cases must be included for *eval* on *Fst* and *Snd*.

The crucial difference between ADTs and GADTs is that GADTs specify the full type of a data constructor, allowing the result type of the constructor to have arbitrary type-parameters for the GADT's type constructor. For example, *Term* can be rewritten as the following on the left-hand side:

$$\begin{array}{ll}
\textbf{data}\ Term\ t\ \textbf{where} & eval :: Term\ a \to a \\
\quad Pair :: Term\ a \to Term\ b \to Term\ (a,b) & eval\ (Pair\ x\ y) = (eval\ x, eval\ y) \\
\quad Fst\ \ :: Term\ (a,b) \to Term\ a & eval\ (Fst\ x) \quad = fst\ (eval\ x) \\
\quad Snd\ \ :: Term\ (a,b) \to Term\ b & eval\ (Snd\ x) \quad = snd\ (eval\ x) \\
\quad Val\ \ :: a \to Term\ a & eval\ (Val\ x) \quad = x
\end{array}$$

The type parameter of *Term* encodes the type of a term as a Haskell type. Because a term type is known, *eval* can be rewritten as on the right-hand side, where a well-typed Haskell value is returned as opposed to the *Val* datatype encoding *Term* values used previously. Furthermore, *eval* cannot be applied to malformed *Term* terms, thus error handling cases are unnecessary. GADTs thus allow a form of lightweight dependent-typing by permitting types to depend on data constructors.

D.3. Type families

Type classes in Haskell fix the types of their methods with type signatures in the class declaration. The *type families* extension to GHC allows types of a class method to vary *per-instance* of a class by defining a family of types associated with the class, indexed by its parameter, and using the family in method signatures [**CKJ05**]. Type families, also called *type-indexed families of types*, thus provide a simple form of type-level functions, evaluated during type checking

[**CKJM05, CKJ05**] Type families describe a number of rewrite rules from types to types, consisting of a *head declaration* specifying the name and arity of the type family and a number of *instance declarations* defining the rewrite rules. For example, the following type family provides a projection function on pair types:

type family $Fst\ t$
type instance $Fst\ (a, b) = a$

Type families are *open* allowing further instances to be defined throughout a program or in another module. Instances of a family are therefore *unordered* thus a application of a family to an argument does not result in ordered pattern matching of the argument against the family's instances. Consequently, to preserve uniqueness of typing, type family instances must not *overlap* or at least must be *confluent* i.e. if there are two possible rewrites for a type family then the rewrites are equivalent.

The rewrite rules of a type family have strict restrictions to ensure that they are confluent and terminating [**SJCS08**]. Type families may be recursive, as long as the size of the type parameters of the recursive call are less than the size of the type parameters in the instance making the recursive call. For example, the following defines an append operation on the type-level list representation:

type family $Append\ x\ y$
type instance $Append\ Nil\ z = z$
type instance $Append\ (Cons\ x\ y)\ z = Cons\ x\ (Append\ y\ z)$

which can be used to type a data-level append function for data values of *List*:

$append :: List\ x \to List\ y \to List\ (Append\ x\ y)$
$append\ Nil\ z = z$
$append\ (Cons\ x\ y)\ z = Cons\ x\ (append\ y\ z)$

Ignoring the **type family** and **instance** keywords it is easier to read this family as recursive function on the data-constructors for *List*, although the family is in fact a type-function.

D.4. Constraint families

Analogous to the concept of a type family is that of *constraint family*, proposed by Orchard and Schrijvers [**OS10**] which allow the constraints of a class method to vary per-instance similarly to type families. They may be written separately from a class, or in associated form. As with type families, the definition of constraint families is open, allowing instances to be defined in other modules that are separately compiled.

The following example from [**OS10**] specifies a polymorphic embedded domain-specific language (in "finally tagless" style [**CKS07**]), where the language is defined via a class, allowing different evaluation semantics for the language based on some type. An associated constraint family allows each instance to vary its constraints:

```
class Expr sem where
  constraint Constr sem a
  const :: Constr sem a ⇒ a → sem a
  add :: Constr sem a ⇒ a → sem a

data E a = E {eval :: a}

instance Expr E where
  constraint Constr E a = Num a
  const c = E c
  add e1 e2 = E $ eval e1 + eval e2
```

The semantics of terms in the *Expr* language is indexed by the *sem* type. Each instance of *Expr* can have its own specialised constraints via an instance of the *Constr* constraint family.

Usefully, an associated constraint family or type family can have a default instance in the class declaration e.g. *Expr* could have been declared:

```
class Expr sem where
  constraint Constr sem a = ()
  ...
```

where () is the empty constraint. Instances of *Expr* could then omit an instance of the *Constr* constraint family. Default instances are useful for backwards compatibility.

Constraint kinds. A recent extension to GHC subsumes the constraint family proposal by redefining constraints as *types* with a distinct *constraint kind*, thus type families may be reused, returning types of kind *Constraint*. The constraint kinds extension,[1] implemented by Bolingbroke [**Bol11**], negates the need for a syntactic and semantic extension to the type checker to add constraint families. Under the extension, a class constructor, *e.g.*, *Ord*, is a type constructor of kind $* \rightarrow$ *Constraint*. All constraints, including equality constraints [**CKJ05**], implicit parameters [**LLMS00**], conjunctions, and empty constraints, are constraint-kinded types, *e.g.*

- $(Show\ a, Ord\ a) :: Constraint$ [conjunction of constraints]
- $() :: Constraint$ [empty constraints]
- $(a \sim Int) :: Constraint$ [equality constraints]
- $(?x :: Int) :: Constraint$ [implicit parameters]

Depending on the context, tuples can be types or constraints *i.e.* $(,) :: * \rightarrow * \rightarrow *$ or $(,) :: Constraint \rightarrow Constraint \rightarrow Constraint$ (conjunction of constraints) and $() :: *$ or $() :: Constraint$ for the unit type or empty (true) constraint.

Previously in Haskell type signatures were of the form $C \Rightarrow \tau$ where C denoted a constraint term and τ a type term. Now, the term appearing on the left of an arrow $\Rightarrow$ is a type term of kind *Constraint*. Given constraints as types, any type system features on type terms can now be reused on constraints. Thus, constraint families can be emulated with type families of *Constraint*-kinded types. For example, the *Expr* class with constraint families can be expressed:

```
class Expr sem where
  type Constr sem a :: Constraint
```

[1] Constraint kinds are enabled by the pragma {-# LANGUAGE ConstraintKinds #-}. At the time of writing it is also necessary to import *GHC.Prim*.

```
  constant :: Constr sem a ⇒ a → sem a
  add :: Constr sem a ⇒ a → sem a


data E a = E { eval :: a }
instance Expr E where
  type Constr E a = Num a
  ...    -- as before
```

The only differences between this definition and the previous are:

1. the use of the **type** keyword, instead of **constraint**, to define *Constr* as a type family;
2. the explicit kind signature on *Constr*, specifying that *Constr* is a type family of constraint-kinded types.

Constr is therefore a type family of kind $* \to * \to Constraint$.

Appendix E

Further example source code

E.1. Common code

E.1.1. Type-level numbers

Type-indexed natural numbers can be encoded in Haskell as follows using a GADT, where the type represents the single value inhabiting the type:

```
data Z
data S n

data Nat n where
  Z :: Nat Z
  S :: Nat n → Nat (S n)
```

Type-level operations on natural numbers can be defined using type families, *e.g.*

```
type family Add n m
type instance Add m Z = m
type instance Add m (S n) = Add (S m) n
```

A useful helper class provides a *less-than* predicate:

```
class LT n m where
instance LT Z (S n) where
instance (LT n m) ⇒ LT (S n) (S m) where
```

Integers. Type-level integers can be defined in terms of *Nat* as either a negative or positive natural number:

```
data Neg n
data Pos n

data IntT n where
  Neg :: Nat (S n) → IntT (Neg (S n))
  Pos :: Nat n → IntT (Pos n)
```

E.1.2. Type-level lists

A type-level list is essentially a heterogeneously-typed list, where the types of the elements is encoded by a type-level lists. A possible definition is the following:

```
data Nil
data Cons x xs
```

data *List t* **where**
  Nil :: *List Nil*
  Cons :: $x \to$ *List xs* $\to$ *List* (*Cons x xs*)

The *List* type thus encodes its element's types in its type parameter t as a type-level list, *e.g.*:

(*Cons* 1 (*Cons* `"hello"` (*Cons* `'a'` *Nil*))) :: (*List* (*Cons Int* (*Cons String* (*Cons Char Nil*))))

The *list membership* predicate can be defined as a type class:

class *Member i ixs*
instance *Member i* (*Cons i ixs*) -- List head matches
instance *Member i ixs* $\Rightarrow$ *Member i* (*Cons i′ ixs*) -- List head does not match, recurse

E.2. Example comonads

Example E.2.1 (product comonad)**.** The *product comonad* is the simplest comonad (after the identity comonad) and can be used to structure notions of implicit, read-only parameters carried along with a computation.

data *Prod x a* = *Prod a x*

instance *Comonad* (*Prod x*) **where**
  current (*Prod a x*) = *a* -- *i.e.* $\epsilon(a, x) = a$
  extend k (*Prod a x*) = *Prod* (*k* (*Prod a x*)) *x* -- *i.e.* $\delta(a, x) = ((a, x), x)$

Example E.2.2 (coproduct of comonads)**.** Given two comonads, a comonad can be formed as the sum of the two.

data *CoProd f g a* = *Either* (*f a*) (*g a*)

instance (*Comonad f*, *Comonad g*) $\Rightarrow$ *Comonad* (*CoProd f g*) **where**
  current (*Left x*) = *current x*
  current (*Right x*) = *current x*
  extend k (*Left x*) = *Left* (*extend* (*k* $\circ$ *Left*) *x*)
  extend k (*Rright x*) = *Right* (*extend* (*k* $\circ$ *Right*) *x*)

Example E.2.3 (suffix lists)**.** *Suffix lists* allow only those contexts that are greater than, or equal to, the current context to be accessible, *e.g.* $\delta\,[1, 2, 3] = [[1, 2, 3], [2, 3], [3]]$ where the cursor is the head of the list, defined:

current $(x : xs) = x$
extend k $[x] = [k\ [x]]$
extend k $(x : xs) = k\ (x : xs) : (\textit{extend}\ k\ xs)$

Alternatively, the notion of a *prefix list comonad* can be defined by taking the current context as the last element of the list, *i.e.*, $\epsilon\ xs = tail\ xs$, and defining accessibility as only those contexts that are less than, or equal to, the current, *e.g.* $\delta\ [1,2,3] = [[1],[1,2],[1,2,3]]$, defined:

```
current x = last x
extend k x = extend' k (reverse x)
  where extend' k [x] = [k [x]]
        extend' k (x : xs) = (extend' k xs) ++ [k (reverse $ x : xs)]
```

Example E.2.4 (rotation lists)**.** *Rotation lists* resemble the fixed lists in Section 3.2.1, where the cursor can be taken as an element in the list and data elements are rotated left through the cursor, *e.g.* taking the cursor as the head element $\delta[1,2,3,4] = [[1,2,3,4],[2,3,4,1],[3,4,1,2],[4,1,2,3]]$, defined:

```
instance Comonad [] where
  current xs = head xs
  extend k xs = snd $ foldl (λ(xs, ys) → λ_ → (lrotate xs, ys ++ [f xs])) (xs, []) xs
                  where lrotate (x : xs) = xs ++ [x]
```

Example E.2.5 (safe pointed-list comonad)**.** From Section 3.2.2.2, using the type-indexed natural numbers data type defined above (Appendix E.1.1):

```
data SList n a where
  Nil :: SList Z a
  Cons :: a → SList n a → SList (S n) a

data PList n a = ∀ m . (LT m n) ⇒ PList (SList n a) (Nat m)
```

where LT tests that the pointer m is less than the length of the list.

A useful helper function converts safe pointed lists to standard lists:

```
toList :: PList' n a → [a]
toList (PList' x _) = toList' x
  where toList' :: SList n a → [a]
        toList' Nil = []
        toList' (Cons x xs) = x : (toList' xs)
```

The *PList* type permits a comonad with a manifest pointer, defined:

```
instance (PCobind Z (S n) (S n)) ⇒ Comonad (PList (S n)) where
  current (PList l n) = coreturn' l n
                          where coreturn' :: (SList p a) → Nat q → a
                                coreturn' (Cons _ xs) (S n) = coreturn' xs n
```

```
                                    coreturn' (Cons x _) Z = x
  extend k (PList l n) = PList (pcobind k l Z) n

class (m~(Add c n)) => PCobind c n m where
  pcobind :: (PList m a -> b) -> SList m a -> Nat c -> SList n b

instance PCobind c Z c where
  pcobind k xs n' = Nil

instance (LT c m, PCobind (S c) n m) => PCobind c (S n) m where
  pcobind k xs n' = Cons (k (PList xs n')) (pcobind k xs (S n'))
```

Example E.2.6 (list zipper comonad)**.** The list zipper data is defined by a path, whose head is taken as the focus element:

```
type LPath a = [a]
data ListZipper a = LZ (LPath a) [a]
```

Operations for zipping (going *left* in a list) and unzipping (going *right* in a list) are defined:

```
left :: ListZipper a -> ListZipper a
left (LZ [] ys) = LZ [] ys
left (LZ (x : xs) ys) = LZ xs (x : ys)

right :: ListZipper a -> ListZipper a
right (LZ xs (y : ys)) = LZ (y : xs) ys
right (LZ xs [])       = LZ xs []
```

Example E.2.7 (composing comonads)**.** Section 3.2.7 showed the construction for composing comonads. This can be encoded in Haskell in the following way:

```
class ComonadDist c d where
  cdist :: c (d a) -> d (c a)

newtype (g : . f) x = O (g (f x))

instance (Comonad c, Comonad d, ComonadDist c d) => Comonad (c : . d) where
  current (O x) = current ∘ current $ x
  disject (O x) = O ∘ (cmap (cmap O)) ∘ (cmap cdist) ∘ (cmap (cmap disject)) ∘ disject $ x
  cmap f (O x) = O (cmap (cmap f) x)
```

Quis est victor tenet prandium.