COMPUTER SCIENCE TRIPOS – PART III

Semantics of temporal type systems

Dmitrij Szamozvancev
Downing College

UNIVERSITY OF CAMBRIDGE

*A dissertation submitted to the University of Cambridge
in partial fulfilment of the requirements for
Computer Science Tripos, Part III*

University of Cambridge
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
UNITED KINGDOM

Email: ds709@cam.ac.uk

May 31, 2018

Declaration

I, Dmitrij Szamozvancev of Downing College, being a candidate for Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11,874

Signed:

Date:

Abstract

Writing interactive programs is challenging for two main reasons: one needs to think about the state of the system over a long period of time, and the source code has to be structured around event-handling constructs instead of the natural flow of the program. Functional reactive programming (FRP) is a declarative paradigm that aims to overcome these problems: the purely functional setting simplifies reasoning about system state, and programs can be composed from time-varying values and events in an intuitive manner. Unfortunately, the elegant semantics of pure FRP do not translate into an efficient implementation, so most practical systems have to find a compromise between ease of use and performance. Moreover, crucial temporal properties such as causality are not enforced by the basic FRP formulation.

We present a functional reactive programming framework with static guarantees of temporal properties and a sound denotational semantics that can be implemented in an efficient way. It combines ideas from recent research into the theoretical basis of FRP such as reactive types, temporal categories and delay operators. We eliminate the need for polling – usually required with an inductive implementation of FRP types – by expressing events as a "value with some delay" packaged into an existential type. Our surface language abstracts away the granular notion of time and lets users handle events and behaviours in a high-level, statically correct way. The system is formalised in Agda, including the typed abstract syntax, equational theory and full categorical semantics.

This work sets the basis for further research that would culminate in an reactive programming framework that retains both the intuitive semantics of pure FRP and the efficiency of continuation passing style implementations.

Acknowledgements

I would like to thank my supervisor, Neel Krishnaswami, for his guidance of my work throughout the duration of this project. He helped me get up to speed with Agda and the practical aspects of language formalisation, which gave me a taste of programming language research. I always left our weekly meetings with a whole host of new ideas and suggestions on how to solve a difficulty I encountered or how to proceed with the implementation. Of course, I have Neel to thank for the project idea as well – it has been really rewarding to work in and contribute to his exciting and active field of research.

I am also grateful to my friends and family for their continuing support and encouragement. Michael Gale was always on hand when I wanted to complain about a particularly ugly Agda error message or express my excitement upon discovering a nice theoretical property of something I've been using for months. My parents provided some much needed emotional support in my moments of frustration – "Muuum, I'm stuck on a proof again" is something they must have become used to hearing by now. I am lucky to be surrounded by a great bunch of people at Downing College, who made my undergraduate years really enjoyable. Finally, I would like to take this opportunity to thank the Cambridge Bursary Scheme and Dr Jan Hruska for supporting my studies during my four years at Cambridge, which would not have been possible without their generosity.

Contents

CHAPTER 1

Introduction

Most programs we encounter in our day-to-day lives are interactive: they are in constant communication with the environment, receiving input and producing output. One of the main challenges in developing such systems is that different components will be executed at different times: we need to reason about the state of our program over a long series of state changes. Moreover, in imperative languages, complex interactive program flow often requires a fragile network of interconnected event producers, callback functions and sequential code.

Functional reactive programming (FRP) is a paradigm which enables writing interactive programs in a high-level, declarative manner (Elliott and Hudak, 1997). Instead of having callbacks and an event loop, we manipulate time-varying values directly, either as continuous signals (behaviours) or discrete occurrences (events). Combinators for switching, sampling, etc. are used to write reactive programs in a natural, intuitive manner, explicitly indicating their dependence on time. The pure nature of the paradigm also means that we do not have to worry about mutable system state over time, and the concurrency model remains simple.

However, FRP is not the perfect solution: the improvement in flexibility and ease of programming is sometimes counteracted by performance issues. The elegant, high-level semantics translate into a suboptimal implementation which suffers from problems such as memory leaks, high latency and unpredictable resource use. Pure FRP also allows programs which violate *causality*, a basic principle of reactive programming: we can describe systems whose output depends on future input. Alternative formulations of FRP may change the underlying implementation or constrain the kinds of programs that can be written to ensure performance and correctness.

One of the main lines of FRP research is developing specialised type systems that statically enforce causality and efficient resource usage. Jeffrey (2012) and Jeltsch (2012) independently discovered a formal connection between linear temporal logic (LTL) and type systems for FRP: LTL propositions can be interpreted as types for FRP programs, with the modalities corresponding to the event and behaviour types. This eliminates non-causal programs – whose types would correspond to invalid temporal propositions – but ignores matters of performance. Specifically, the usual inductive-coinductive definition of possibility corresponds to a rather inefficient event type: handling an event would require constant polling, which is wasteful for most interactive applications.

1.1 Project description

This project aims to develop a type system based on temporal logic, along with a denotational semantics that allows for an efficient implementation of a reactive programming language. The type system is based on the judgemental modal logic of Pfenning and Davies (2001) and higher-order FRP language of Krishnaswami (2013): its types and constructs enforce the correctness properties of temporal logic such as causality and inaccessibility of non-persistent values. The language is interpreted in the category of time-dependent types utilising a novel definition of the event modality which uses Krishnaswami's delay operator. This formulation of events avoids any induction or recursion which would have to be handled by constant polling. In the future, we plan to show that the definition can be translated to continuation passing style as suggested by Paykin, Krishnaswami, and Zdancewic (2016), thereby combining the intuitive semantics of FRP with an efficient implementation.

Our main contributions are the following:

- An Agda formalisation of Pfenning and Davies' judgemental modal logic with proof terms, explicit substitutions and equational theory.
- A concrete model of LTL in the category of reactive types with an adjunction-induced comonad $\square$ and non-inductive $\square$-strong monad $\diamond$.
- Machine-checked, sound categorical semantics of the syntax in our reactive category.

CHAPTER 2

Background

This section introduces the context of our research: the relationship of temporal logic and functional reactive programming.

2.1 Modal and temporal logics

Modal logic is an extension of classical propositional and predicate logic with truth qualifiers called *modalities*: operators that can be applied to a logical statement in order to change its mode of truth. The two basic modalities are *necessity* and *possibility*: for example, for any logical statement S, $\Box S$ means "S is necessarily true", and $\Diamond S$ means "S is possibly true". These general modalities can be specialised to concrete fields of study, such as time (*always*, *eventually*), duty (*obligatory*, *permissible*) and provability (*provable*).

Despite the different linguistic interpretations, most subtypes of modal logic can be given a general semantics based on possible worlds and a transition relation (Kripke semantics), or described in an axiomatic framework. In the latter case, we extend the Hilbert-style proof system for propositional logic with a dual pair of (interderivable) modal operators and two axioms:

- $\neg\Diamond A = \Box\neg A$, $\neg\Box A = \Diamond\neg A$
- **N** (necessity): if P is a theorem, then so is $\Box P$
- **K** (distribution): $\Box(P \longrightarrow Q) \longrightarrow (\Box P \longrightarrow \Box Q)$

This is the most basic type of modal logic, known as K – we get other types by adding new axioms expressing the desired properties of our domain. For example, axiom $\mathbf{T} = \Box P \longrightarrow P$ (reflexivity) states that if P is necessarily true then P is also the case; axiom $\mathbf{4} = \Box P \longrightarrow \Box\Box P$ (transitivity) states that necessity of P implies the necessity that P always holds. Adding these axioms to K brings us to S4 modal logic, which can be interpreted as a model of time.

Temporal logic is the logic of time-dependent propositions, and its basic modalities are *always* ($\Box$) and *eventually* ($\Diamond$). The classic example of a time-dependent statement is P = "It is raining." – clearly, this is not universally true, but it can change from time to time. $\Box P$ then means "It is always raining", while $\Diamond P$ asserts that "It will eventually rain." Modalities can be iterated and sequenced, as they are part of the syntax; for example, $\Diamond\Box P$ states that "It will always be the case that it will eventually rain" (i.e. it will rain infinitely often). A more general modality is *until* ($\mathcal{U}$): $P \mathcal{U} Q$ holds if Q eventually holds and P holds at least until then. Both necessity and possibility can be derived from $\mathcal{U}$: $\Diamond P = \top \mathcal{U} P$ and $\Box P = P \mathcal{U} \bot$.

Temporal logic is widely employed in the field of *formal specification*, where it is used to express the requirements of a system in a rigorous and verifiable way (Pnueli, 1977). Anytime we want to state a property of a system that involves time (either implicitly or explicitly), we may express it as a temporal statement. For example, "The payment confirmation screen must only be shown after the transaction has been completed" or "Once the transaction is completed, a confirmation email should be sent to the user's account." Verifying that a system fits a given specification is the task of *model checking*, which is an important field of research in diverse fields such as formal systems, hardware design and security protocols.

2.2 Functional reactive programming

2.2.1 Main principles

Functional reactive programming (FRP) is a paradigm for asynchronous dataflow programming, centred around the manipulation of time-dependent values such as signals and events. It was introduced by Elliott and Hudak (1997) for the purpose of describing animations in a high-level, declarative way, specifying *what* the animation should look like instead of *how* it must be implemented. Elliott and Hudak encompass the core ideas of FRP in the following principles:

Temporal modelling The main primitives of FRP are time-dependent values called *behaviours* or *signals*. The value of behaviours varies continuously with time, so they can naturally represent things like velocity, mouse position, value streams, etc.

Event modelling FRP events mark the occurrence of something that should change the behaviour of a program. Examples of events are mouse clicks, collisions, temporal predicates (such as "every 5 seconds"), etc.

Declarative reactivity Reactivity of FRP programs comes from changing some behaviour in response to an event occurring. This should be expressed declaratively, in terms of temporal composition instead of state changes.

Denotational design (Elliott, 2009a) The emphasis should be put on the meaning of a program instead of its implementation, and the meaning of a program should derive from the meaning of its components.

Since Elliot and Hudak introduced these ideas, the functional reactive paradigm has been applied to many other domains including music (Quick and Hudak, 2013), robotics (Hudak, Courtney, et al., 2002) and graphical user interfaces (Czaplicki and Chong, 2013). However, the various implementations (usually as libraries) often deviate from the original principles of FRP (Elliott, 2015). The reason for the differences is that mathematical elegance does not imply efficient implementation, and this is a compromise that has long plagued the FRP community. The next section explores this tension and some proposed solutions.

2.2.2 Implementations

Functional reactive programming systems have two main implementations (Jeltsch, 2011): *pull-based* or demand-driven, and *push-based* or data-driven.

Pull-based implementations treat behaviours as streams of values, usually expressed as functions of time. The primary advantage of this approach is that it is corresponds well to the denotational semantics of time-dependent behaviours: for example, the "meaning" of an animation is a time-dependent image, and it is implemented as a function of type $Time \longrightarrow Image$. This makes it easy to define various FRP combinators, and simplifies the development of even large reactive systems.

However, pull-based FRP suffers from significant performance issues. The consumers must regularly poll the streams to react to changes; while this is effective for continuously changing values (such as mouse position), it is very wasteful for infrequent events (such as a button press). At the same time, the size of the polling interval imposes latency on the system: it cannot react to events instantaneously, only within one half of the polling interval on average.

Push-based implementations are at the opposite end of the spectrum: they are efficient, but sacrifice a lot of the denotational elegance and ease of use that pull-based FRP provides. Instead of composing continuously varying behaviours and reacting to events, we use callback functions to handle events as soon as they happen, without needing to sample the value streams. For large systems, this can easily result in the notorious "callback hell", a common issue in asynchronous reactive programming.

There have been several attempts to address these issues. *Push-pull FRP* (Elliott, 2009b) combines the advantages of both methods by treating discrete and continuous behaviours separately. *Real-time* and *event-driven FRP* (Wan, Taha, and Hudak, 2002) is based on an operational semantics for a standalone FRP language which has explicit time- and space guarantees – this makes the implementation suitable for real-time and embedded systems. *Arrowised FRP* (Nilsson, Courtney, and Peterson, 2002) tackles space leaks, which occur when signals accumulate all past values unnecessarily. In AFRP, programs are composed from *signal functions* instead of signals, which also fit the *arrow* abstraction of Hughes (2000) and therefore benefit from generic combinators and a special syntactic treatment in Haskell (Paterson, 2001).

A considerable part of current FRP research is on developing new libraries or improving existing FRP implementations. However, a lot of work is done on exploring the logical and type-theoretic foundations of FRP: what theoretical features correspond to what desired properties of the implementation (e.g. avoiding spacetime leaks, fairness, causality, type safety, etc.). Again, we encounter the tension between theoretical models and practical implementations: many formal languages developed in FRP research propose or rely on features which are hard to implement efficiently.

2.3 Temporal Curry–Howard correspondence

There is a common thread in the fields of temporal logic and functional reactive programming: manipulating or reasoning about *time*. The two modalities of temporal logic (*always* and *eventually*), and the two first-class primitives of FRP (*behaviours* and *events*) also exhibit a clear connection: behaviours *always* have values, while events *eventually* have a value. These parallels suggest a clear connection between the two concepts, and in the field of type theory this really hints at a Curry–Howard correspondence between temporal logic and FRP. Indeed, as discovered independently by Jeffrey (2012) and Jeltsch (2012), linear temporal logic can be developed into a type system for discrete-time functional reactive programming: $\Box$ becomes the behaviour type constructor, while $\Diamond$ can be used to construct events.

Jeffrey and Jeltsch propose the same basic idea: using not only reactive values, but *reactive types* as well, i.e. type families indexed by discrete set of times. Jeffrey uses this foundation to build a dependently typed FRP framework where various temporal properties (such as causality or decoupling) are enforced in the types. Jeltsch develops a common categorical semantics for LTL and FRP, extending it with other features such as the $\mathcal{U}$ modality or processes.

2.4 Agda formalisation

All developments in this dissertation have been formalised in Agda (Norell, 2008), a dependently typed programming language and proof assistant. We do not have space to describe the implementation in great detail here; however, we do mention interesting aspects of the code occasionally. One convenient feature of the language is its flexible Haskell-like syntax that supports Unicode characters – this way, a lot of Agda code looks just like the mathematics it describes. We take advantage of this syntactic freedom and use “pseudo-Agda” throughout the dissertation, avoiding distracting implementation details like implicit arguments and function extensionality, and formatting categorical expressions as normal mathematics instead of Agda code. We believe this is sufficient to convey the main idea of the implementation without getting bogged down in the details.

All code for this project was written by hand[1], including a custom formalisation of basic category theory which provided the basis of our semantics. The only external library used (apart from the standard one) was Holes[2] by Bradley Hardy (2017), which simplifies applying congruence rules in equational proofs.

In the next section, we start the discussion of our reactive language.

[1]`https://bit.ly/2JgEaH9`
[2]`https://github.com/bch29/agda-holes`

CHAPTER 3

Syntax

We begin our investigation of the semantics of temporal type systems by introducing a simple language for reactive programming. This chapter presents the syntactic aspects of the language: the type system, term grammar, substitution lemmas and term equality.

3.1 Types and terms

The core of our language is the simply typed lambda calculus extended with with products, sums, and additional types and constructs which support reactive programming. Importantly, the notion of time is exposed in a limited way: every type is annotated with a *temporal qualifier*, which denotes whether the expression of a reactive type is always well-typed, or just at the current time step. This high-level treatment of time means that the temporal semantics involving discrete time steps and delays are kept abstract. It also gives us static guarantees that values are only used when they are available: for example, when handling future events, we cannot refer to variables only accessible in the present. Hence, the temporal properties of our domain are encoded in the type system, resulting in language for safe reactive programming.

This language is mainly based on the constructive modal logic of Pfenning and Davies (2001), who describe a type theory based on a system of modal judgements in the style of Per Martin-Löf (1998). In particular, we retain the distinction between true and valid judgements (which translate into the now and always temporal qualifiers), and proof terms and proof expressions (which become terms and computations). The temporal qualifiers were inspired by Krishnaswami (2013), who also used a later qualifier for values available on the next time step. As our aim is to hide the "granularity" of time, neither this qualifier, nor the one-step delay term δ appears in our syntax.

3.1.1 Types and contexts

Types The types of our language (Fig. 3.1) are standard unit, product, sum and function types. In addition, we have two top-level types for *events* and *stable types*: these will correspond to the temporal modalities in our denotational semantics. Stable types are inhabited by values which do not change over time and are always available – these are the non-reactive entities of our language. Events correspond to the usual event type constructor in FRP and represent values that become available at some future time.

Qualifiers The type language is supplemented by two *temporal qualifiers* (Fig. 3.2), which denote the time when a term of a given type is available. Following Martin-Löf's terminology, a type with a temporal qualifier forms a *judgement*, which in a constructive setting is called a *typing judgement.* A term of *reactive type A* now can be used as a term of type A at the present, but it may not be available in the future; in particular, a variable available now is inaccessible when an event happens. On the other hand, a term of *persistent type A* always will remain available for all future times.

It is worth comparing this formulation the one described by Krishnaswami (2013). The $\square$ type constructor and stable qualifier are analogous to our persistent types. However, we replace the discrete next-step operator $\bullet$ with a more abstract Event type which does not explicitly mention time steps and delays. This lets us talk about future occurrences in the most general way possible: an event happens at some time in the future, and a stable type is guaranteed to be available at that time. As an important extension, we plan to adapt Krishnaswami's guarded temporal recursive type $\hat{\mu}\alpha.\, A$ in our syntax, using Event as the guarding operator instead of $\bullet$. This will allow us to define more complex FRP constructs such as streams, processes and resumptions ensuring that they are temporally well-founded. For example, streams (time-dependent values) can be encoded as Stream $A = \hat{\mu}\alpha.\, A \times \alpha$, which expands to $A \times \text{Event}\ (A \times \text{Event}\ (A \times\ \ldots))$, expressing that a stream is a sequence of values separated by some delays. As events are primitives in our language, we do not need to define them coinductively – this avoids the main source of inefficiency of Krishnaswami's system, the need for recursive polling to react to events. As an alternative, we will also consider the fair reactive programming system of Cave et al. (2014), who adopt separate least and greatest fixed points which can express liveness properties of programs statically (such as distinguishing normal and weak eventuality).

Contexts As is common in type theory and logic, we give the types of free variables in open terms in a *typing environment* or *context*: for example, $\Gamma \vdash M : A$ now means that in the context Γ, the term M has type A at the current time. Environments are implemented as a list of judgements, so each free variable has a temporally qualified type – this is crucial in the definition of the typing rules, as it lets us restrict the availability of variables in subterms. An important operation on contexts is *stabilisation*: removing all variables with the now qualifier, leaving only ones that are always available. For instance, the term $x : A \times B$ now $\vdash$ fst $x : A$ now is well-typed, but $(x : A \times B \text{ now})^s \vdash$ fst $x : A$ now is not, as the stabilisation discards the reactive variable x from the context.

3.1.2 Terms and typing rules

The terms of our system also derive from the simply typed lambda calculus, extended with the modal proof terms and expressions of Pfenning and Davies (Fig. 3.1). These entities are translated into two mutually defined syntactic constructs called *terms* and *computations*: terms are used to write programs for the current time step, while computations represent code that runs when some future event is encountered. Every construct has an accompanying typing rule, shown on Figs. 3.3 and 3.4.

Basic terms The lambda calculus, product and sum terms are all standard so we will not detail them here – more interesting are the terms dealing with reactivity. The extract M term expresses the intuitive fact that if a type is always available, its value can be extracted at any particular time. On the other hand, a term of persistent type can only be constructed from a term which does not depend on reactive variables – thus, the term persist M has type A always only if M has type A now in the stabilised context.

Stable types The stable type introduction and elimination terms express the inherent similarity between the Stable type constructor and the always qualifier: both involve reasoning about global time instead of just the present. The stable term turns any persistent type A always into a reactive type Stable A now, and the stable binding let stable $s = M$ in N lets us use a stable term M : Stable A now in a body N as a persistent type variable s. Note that we cannot simply convert Stable A now to A always, as that would not result in a sound and complete system (see Pfenning and Davies, 2001, Section 4).

Events While stable types correspond to the always temporal qualifier, events are externalised as a new kind of expression called a *computation*. The reason for this asymmetry is that necessity (stable types) is more useful as an assumption, while possibility (events) usually appears as a consequent. The judgement $\Gamma \vdash C \div A$ now states that C is a computation that will produce a term of type A now in the future. All such computations $C \div A$ now can be converted into normal terms of type Event A now using the event constructor, and every normal term $M : A\ q$ can be turned into the trivial computation **pure** $M \div A\ q$ which returns M immediately. These two constructs let us move between the "realms" of terms and computations, but not without explicitly enforcing that we cannot look into the future: a computation $C \div A$ now (which might only return a term tomorrow) can never be used as a term available today.

Given a term M: Event A now, we have no way of extracting its value directly, as it may not be available in the present. But we are allowed to assume that the event will occur at some future time and handle its returned value of type A now at that point. The binding **let evt** $x = M$ **in** C lets us use this future value in a computation $C \div B$ now – however, the handler is going to be run in the future, so it may not have access to the reactive variables present now. Thus, we may only use persistent variables in the handler, i.e. it must be well-typed in the stabilised context. Computations and events prevent us from looking into the future, while stabilisation forces us to forget about the past.

We need to consider how computations interact with stable types: they should be available at all times, yet our current stable binding only allows us to use them in the present. We therefore add a version of destructor **let stable** $s = M$ **in** C that binds a stable type in the computation C.

The last expression in our grammar allow us to "race" two events and run different code based which one happens first. Whereas in nested event handlers, stabilisation removes the outer event from the context of the inner handler, the select term makes both the value of the occurred event and the "remainder" of the second event available. With the computation **select** M **as** $x \mapsto C_1 \parallel N$ **as** $y \mapsto C_2 \parallel$ **both as** $x, y \mapsto C_3$, we specify three different handlers: C_1 runs if M happens first, C_2 runs if N happens first, and C_3 runs if both events occur at the same time.

Types

$$A, B ::= \mathsf{Unit} \mid A \times B \mid A + B \mid A \to B \mid \mathsf{Stable}\, A \mid \mathsf{Event}\, A$$

Terms

$$\begin{aligned} M, N ::=\ & x \mid \lambda x.\, M \mid M\, N \\ \mid\ & \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid [M, N] \\ \mid\ & \mathsf{inl}\, M \mid \mathsf{inr}\, M \mid \mathsf{case}\, M\ \mathsf{of}\ \mathsf{inl}\, x \mapsto N_1 \parallel \mathsf{inr}\, y \mapsto N_2 \\ \mid\ & \mathsf{extract}\, M \mid \mathsf{persist}\, M \mid \mathsf{stable}\, M \\ \mid\ & \mathsf{let\ stable}\ s = M\ \mathsf{in}\ N \mid \mathsf{event}\, C \end{aligned}$$

Computations

$$\begin{aligned} C, D ::=\ & \mathbf{pure}\, M \mid \mathbf{let\ stable}\ x = M\ \mathbf{in}\ C \mid \mathbf{let\ evt}\ x = M\ \mathbf{in}\ C \\ \mid\ & \mathbf{select}\ M\ \mathbf{as}\ x \mapsto C_1 \parallel N\ \mathbf{as}\ y \mapsto C_2 \parallel \mathbf{both\ as}\ x, y \mapsto C_3 \end{aligned}$$

Figure 3.1 – Syntax of types, terms and computations.

Qualifiers

$$q ::= \mathsf{now} \mid \mathsf{always}$$

Contexts

$$\Gamma, \Delta ::= \cdot \mid \Gamma, x : A\, q$$

Context stabilisation

$$\begin{aligned} (\cdot)^{\mathsf{s}} &= \cdot \\ (\Gamma, x : A\ \mathsf{now})^{\mathsf{s}} &= \Gamma^{\mathsf{s}} \\ (\Gamma, x : A\ \mathsf{always})^{\mathsf{s}} &= \Gamma^{\mathsf{s}}, x : A\ \mathsf{always} \end{aligned}$$

Figure 3.2 – Judgements, contexts and stabilisation.

Term typing rules

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\text{var}) \qquad \frac{\Gamma \vdash M : A\ \mathsf{now}}{\Gamma \vdash \mathsf{inl}\, M : A + B\ \mathsf{now}}\ (\text{inl})$$

$$\frac{\Gamma, x : A\ \mathsf{now} \vdash M : B\ \mathsf{now}}{\Gamma \vdash \lambda x.\, M : A \to B\ \mathsf{now}}\ (\text{lam}) \qquad \frac{\Gamma \vdash M : B\ \mathsf{now}}{\Gamma \vdash \mathsf{inr}\, M : A + B\ \mathsf{now}}\ (\text{inr})$$

$$\frac{\begin{array}{c}\Gamma \vdash M : A \to B\ \mathsf{now} \\ \Gamma \vdash N : A\ \mathsf{now}\end{array}}{\Gamma \vdash M\, N : B\ \mathsf{now}}\ (\text{app}) \qquad \frac{\begin{array}{c}\Gamma \vdash M : A + B\ \mathsf{now} \\ \Gamma, x : A \vdash N_1 : C\ \mathsf{now} \\ \Gamma, y : B \vdash N_2 : C\ \mathsf{now}\end{array}}{\mathsf{case}\, M\ \mathsf{of}\ \mathsf{inl}\, x \mapsto N_1 \parallel \mathsf{inr}\, y \mapsto N_2}\ (\text{case})$$

$$\frac{\Gamma \vdash M : A \times B\ \mathsf{now}}{\Gamma \vdash \mathsf{fst}\, M : A\ \mathsf{now}}\ (\text{fst}) \qquad \frac{\Gamma \vdash M : A\ \mathsf{always}}{\Gamma \vdash \mathsf{extract}\, M : A\ \mathsf{now}}\ (\text{extract})$$

$$\frac{\Gamma \vdash M : A \times B\ \mathsf{now}}{\Gamma \vdash \mathsf{snd}\, M : B\ \mathsf{now}}\ (\text{snd}) \qquad \frac{\Gamma^{\mathsf{s}} \vdash M : A\ \mathsf{now}}{\Gamma \vdash \mathsf{persist}\, M : A\ \mathsf{always}}\ (\text{persist})$$

$$\frac{\Gamma \vdash M : A\ \mathsf{now} \qquad \Gamma \vdash N : B\ \mathsf{now}}{\Gamma \vdash [M, N] : A \times B\ \mathsf{now}}\ (\text{pair})$$

Figure 3.3 – Term typing rules.

Term typing rules (continued)

$$\frac{\Gamma \vdash M : A \ \mathsf{always}}{\Gamma \vdash \mathsf{stable}\ M : \mathsf{Stable}\ A\ \mathsf{now}}\ \text{(stable)} \qquad \frac{\Gamma \vdash M \div A\ \mathsf{now}}{\Gamma \vdash \mathsf{event}\ M : \mathsf{Event}\ A\ \mathsf{now}}\ \text{(evt)}$$

$$\frac{\Gamma \vdash M : \mathsf{Stable}\ A\ \mathsf{now} \qquad \Gamma, s : A\ \mathsf{always} \vdash N : B\ \mathsf{now}}{\Gamma \vdash \mathsf{let\ stable}\ s = M\ \mathsf{in}\ N : B\ \mathsf{now}}\ \text{(letsta)}$$

Computation typing rules

$$\frac{\Gamma \vdash M : A\ q}{\Gamma \vdash \textbf{pure}\ M \div A\ q}\ \text{(pure)}$$

$$\frac{\Gamma \vdash M : \mathsf{Stable}\ A\ \mathsf{now} \qquad \Gamma, s : A\ \mathsf{always} \vdash C \div B\ \mathsf{now}}{\Gamma \vdash \textbf{let stable}\ s = M\ \textbf{in}\ C \div B\ \mathsf{now}}\ \text{(letsta')}$$

$$\frac{\Gamma \vdash M : \mathsf{Event}\ A\ \mathsf{now} \qquad \Gamma^{\mathsf{s}}, x : A\ \mathsf{now} \vdash C \div B\ \mathsf{now}}{\Gamma \vdash \textbf{let evt}\ x = M\ \textbf{in}\ C \div B\ \mathsf{now}}\ \text{(letevt)}$$

$$\frac{\begin{array}{ll} \Gamma \vdash M_1 : \mathsf{Event}\ A\ \mathsf{now} & \Gamma^{\mathsf{s}}, x : A\ \mathsf{now}, y : \mathsf{Event}\ B\ \mathsf{now} \vdash C_1 \div C\ \mathsf{now} \\ \Gamma \vdash M_2 : \mathsf{Event}\ B\ \mathsf{now} & \Gamma^{\mathsf{s}}, x : \mathsf{Event}\ A\ \mathsf{now}, y : B\ \mathsf{now} \vdash C_2 \div C\ \mathsf{now} \\ \multicolumn{2}{c}{\Gamma^{\mathsf{s}}, x : A\ \mathsf{now}, y : B\ \mathsf{now} \vdash C_3 \div C\ \mathsf{now}} \end{array}}{\Gamma \vdash \textbf{select}\ M\ \textbf{as}\ x \mapsto C_1 \parallel N\ \textbf{as}\ y \mapsto C_2 \parallel \textbf{both as}\ x, y \mapsto C_3 \div C\ \mathsf{now}}\ \text{(select)}$$

Figure 3.4 – Term and computation typing rules.

For example, we can race the events E_1 = "A browser download finishes" and E_2 = "The user closes the browser". If E_2 happens first, we might want to display a confirmation window, but keep waiting for E_2 and close the browser automatically as soon as it happens. If E_1 happens first, we might discard the pending event E_2 and close the browser ourselves.

Examples We now give some example terms of our language, using the syntax defined above. They mainly demonstrate the use of the temporal operators and binding constructs; by adding other types and operations (such as Booleans and conditionals), we can imagine using these in practical reactive programs as well.

$$\begin{array}{ll} \mathsf{current} & : \mathsf{Stable}\ A \longrightarrow A\ \mathsf{now} \\ \mathsf{current} & = \lambda x.\ \mathsf{let\ stable}\ s = x\ \mathsf{in}\ (\mathsf{extract}\ s) \end{array}$$

$$\begin{array}{ll} \mathsf{joinEvts} & : \mathsf{Event}\ (\mathsf{Event}\ A) \longrightarrow \mathsf{Event}\ A\ \mathsf{now} \\ \mathsf{joinEvts} & = \lambda x.\ \mathsf{event}\ (\textbf{let evt}\ ee = x\ \textbf{in}\ (\textbf{let evt}\ e = ee\ \textbf{in pure}\ e)) \end{array}$$

$$\begin{aligned}
&\mathsf{handleEvt} : \mathsf{Event}\ A \longrightarrow \mathsf{Stable}\ (A \longrightarrow \mathsf{Event}\ B) \longrightarrow \mathsf{Event}\ B\ \mathsf{now}\\
&\mathsf{handleEvt} = \lambda x.\, \lambda y.\, \mathsf{let\ stable}\ f_s = y\ \mathsf{in}\\
&\qquad\qquad \mathsf{event}\ (\mathbf{let\ evt}\ e = x\ \mathbf{in}\ (\mathbf{let\ evt}\ e' = \mathsf{extract}\ f_s\ e\ \mathbf{in\ pure}\ e'))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{sampleAt} : \mathsf{Stable}\ A \longrightarrow \mathsf{Event}\ B \longrightarrow \mathsf{Event}\ (\mathsf{Stable}\ A \times B)\ \mathsf{now}\\
&\mathsf{sampleAt} = \lambda x.\, \lambda y.\, \mathsf{event}\ (\mathbf{let\ stable}\ s = x\ \mathbf{in\ let\ evt}\ e = y\ \mathbf{in}\ (\mathbf{pure}\ [s, e]))
\end{aligned}$$

Below are two incorrect specifications for event handling: thanks to the typing rules, there is no term that can inhabit these types.

$$\begin{aligned}
&\mathsf{wrong}_1 : \mathsf{Event}\ A \longrightarrow (A \longrightarrow \mathsf{Event}\ B) \longrightarrow \mathsf{Event}\ B\ \mathsf{now}\\
&\mathsf{wrong}_1 = \lambda x.\, \lambda y.\, \mathsf{event}\ (\mathbf{let\ evt}\ e = x\ \mathbf{in}\ ???) \qquad \text{(context is stabilised so } y \text{ is not in scope)}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{wrong}_2 : \mathsf{Event}\ A \longrightarrow \mathsf{Stable}\ (A \longrightarrow \mathsf{Event}\ B) \longrightarrow B\ \mathsf{now}\\
&\mathsf{wrong}_2 = \lambda x.\, \lambda y.\, (\mathsf{let\ stable}\ s = y\ \mathsf{in}\ ???) \qquad \text{(cannot bind the event } x \text{ in the present)}
\end{aligned}$$

3.1.3 Agda formalisation

Describing formal calculi and programming languages is often quite cumbersome with standard functional languages such as Haskell, especially when it comes to the treatment of variables and binding. However, dependently typed languages with *inductive families* (Dybjer, 1994) make it possible to encode terms and type systems in a very idiomatic, type- and scope-safe way, eliminating the issues of naming and α-conversion via a typed de Bruijn representation of variables (de Bruijn, 1972). Here we give a brief overview of the approach commonly used in Agda, which stems from the developments of Bird and Paterson (1999), Bellegarde and Hook (1994), and Altenkirch and Reus (1999).

Types and contexts The type and judgement representations are standard algebraic datatypes, following the formal grammar of Fig. 3.1. Agda's flexible mixfix operator declarations allow us to very closely model the syntax of the language: for example, judgements are postfix operators acting on a whole type:

```
data Judgement : Set where
    _now    : Type -> Judgement
    _always : Type -> Judgement
```

Contexts are simple snoc-lists of judgements: as we are working with de Bruijn indices, we do not need explicit (variable, type) pairs.

```
data Context : Set where
  ·   : Context
  _,_ : Context -> Judgement -> Context
```

Variables We can define an inductive type for context membership: the type $A \in \Gamma$ is inhabited by a proof that the judgement A appears in the context Γ. We can have two cases: either A is the top (last) element of the context, or it is somewhere in the tail.

```
data _∈_ : Judgement -> Context -> Set where
  top : ∀{Γ A}   -> A ∈ Γ , A
  pop : ∀{Γ A B} -> A ∈ Γ -> A ∈ Γ , B
```

In fact, this is exactly what we need to represent variables. The proof of context membership is an index into the typing environment, and the environment gets extended in every term involving a binder, so this proof will be the de Bruijn index of the variable. Moreover, the indexed representation means that (1) the index can never point out of the context, and (2) the variable is statically typed.

Context operations Context operations are implemented as normal list transformations. Stabilisation is simply a filtering based on the qualifier of the top judgement:

```
_ˢ : Context -> Context
·ˢ                = ·
(Γ , A now)    ˢ  = Γ ˢ
(Γ , A always)ˢ   = Γ ˢ , A always
```

An important predicate on contexts is the *subcontext relation*, $\Gamma \subseteq \Delta$. This predicate can be defined as an *order-preserving embedding* (Chapman, 2009, Section 4.2), an inductive implementation of the category of weakenings described by Altenkirch, Hofmann, and Streicher (1995). The embedding can be seen as a proof that Δ can be transformed into Γ by dropping some elements and keeping others, without changing the order.

```
data _⊆_ : Context -> Context -> Set where
  refl  : ∀{Γ} -> Γ ⊆ Γ
  keep : ∀{Γ Γ′ A} -> Γ ⊆ Γ′ -> Γ , A ⊆ Γ′ , A
  drop : ∀{Γ Γ′ A} -> Γ ⊆ Γ′ -> Γ     ⊆ Γ′ , A
```

There are several lemmas we can prove about the subcontext predicate: for example, that the element predicate is "monotone" and a stabilised context can be embedded into the full context.

```
∈-⊆-mono : ∀{Γ Γ′ A} -> Γ ⊆ Γ′ -> A ∈ Γ -> A ∈ Γ′
Γˢ⊆Γ : ∀Γ -> Γ ˢ ⊆ Γ
```

Terms The implementation of formal languages often follows their grammar definition: we define a datatype for types, a datatype for (untyped) terms, then some sort of (value-level) predicate or relation to determine which terms are well-typed. In languages with inductive families, we instead identify terms with the typing rules, thereby enforcing that any term that we construct is guaranteed to be well-typed.

Terms are therefore implemented as constructors of the "well-typed term" predicate $\vdash$, and "well-typed computation" predicate $\vDash$ (which is used instead of the $\div$ judgement). Agda's syntactic flexibility actually allows us to translate the typing rules almost directly, with the type of each constructor being the associated natural deduction rule (formatted with some line breaks and comments). Below we show a few examples.

```
data _⊢_ : Context -> Judgement -> Set where

  var : ∀{Γ A}       ->                 A ∈ Γ
                                       -------
                     ->                 Γ ⊢ A

  lam : ∀{Γ A B}     ->           Γ , A now ⊢ B now
                                 -------------------
                     ->             Γ ⊢ A => B now

  _$_ : ∀{Γ A B}     ->       Γ ⊢ A => B now  ->   Γ ⊢ A now
                             ---------------------------------
                     ->                Γ ⊢ B now

  persist : ∀{Γ A}   ->              Γ ˢ ⊢ A now
                                    --------------
                     ->             Γ ⊢ A always

  event : ∀{Γ A}     ->                Γ ⊨ A now
                                    ----------------
                     ->           Γ ⊢ Event A now

data _⊨_ : Context -> Judgement -> Set where

  letEvt_In_ : ∀{Γ A B} ->   Γ ⊢ Event A now -> Γˢ , A now ⊨ B now
                            -----------------------------------------
                        ->                    Γ ⊨ B now
```

3.1.4 Substitution

An important metatheoretic operation in any language formalism involving binders is *substitution*, which forms the basis of β-reduction and universal instantiation. It is usually an operation of type $\Gamma \vdash M : A \longrightarrow \Gamma, x : A \vdash N : B \longrightarrow \Gamma \vdash [M/x]N : B$, expressing the substitution of the term M for every free occurrence of the variable x in N. Though it is easy enough to describe informally, formalising the process is quite tricky, as we need to check for variable equality, handle renaming, avoid variable capture, and do everything as efficiently as possible. Fortunately, our formalisation lets us implement *explicit substitutions* (Abadi et al., 1991), which turn substitution into a new syntactic entity σ : `Subst` Γ Δ representing arbitrary context transformations $\Gamma \longrightarrow \Delta$, instead of an ad-hoc operation defined on terms. The main advantage of this approach is that the context transformations can be defined via combinators such as weakening ($^{+}$: `Subst` Γ Δ $\longrightarrow$ `Subst` Γ (Δ, A)), lifting ($\uparrow$: `Subst` Γ Δ $\longrightarrow$ `Subst` (Γ, A) (Δ, A)), stabilisation ($\downarrow^{s}$: `Subst` Γ Δ $\longrightarrow$ `Subst` (Γ^{s}) (Δ^{s})), identity and composition of substitutions; moreover, this can be done without referring to the term language at all.

We combine explicit substitutions with the type-preserving traversal idea of McBride (2005). It is based on the observation that variable renaming and substitution can be expressed as instances of a single term traversal operation, which applies a given function to every free variable of a term. Depending on the return type of this function (either variables or terms), we get generic type-preserving renaming or substitution operations. Defining the traversal function is possible only if the return type supports a set of operations such as mapping to terms and weakening – these are collected into a *syntactic kit*. We adapted the Agda implementation described by Keller (2008) to work with our reactive language; surprisingly, this only required adding one extra kit property that is used to define the $\downarrow^{s}$ combinator for context stabilisation.

Using this syntactic framework we give a generic term and computation traversal function which applies an explicit substitution to every variable of a term. Then, by defining custom context transformations for weakening or substitution, we can specialise the traversal function to any structural or substitution lemma. Below are two examples, given for demonstration purposes – the details of the implementation are outside the main theme of this dissertation, but given that they were a considerable part of the practical work done, we decided to include a more comprehensive discussion in Appendix B for completeness.

```
substitute : ∀{Γ Δ A} -> Subst Γ Δ -> Γ ⊢ A -> Δ ⊢ A
ex-topₛ   : ∀{Γ A B} -> Subst (Γ, A, B) (Γ, B, A)
sub-topₛ : ∀{Γ A} -> Γ ⊢ A -> Subst (Γ, A) Γ

exchange : ∀{Γ A B C} ->   Γ, A, B ⊢ C
                          -------------
                     ->   Γ, B, A ⊢ C
exchange = substitute ex-topₛ

[_/] : ∀{Γ A B}  ->  Γ ⊢ A   ->   Γ, A ⊢ B
                    -----------------------
                 ->         Γ ⊢ B
[M /] = substitute (sub-topₛ M)
```

3.2 Term equality

One characteristic feature of lambda calculi and other computational systems is *term equality*, a relation expressing when two terms of the same type are equal. It is closely related to the property of *soundness* (see Section 5.2): if two terms are equal in the syntax, they must have equal interpretations. This ensures that whenever we establish the equality of two terms, they really do have the same meaning. In this section we present the equational system of our language, combining the usual $\beta\eta$-equality rules of the λ-calculus with the modal reduction and expansion properties of the Pfenning and Davies system.

3.2.1 Reduction rules

Reduction rules are responsible for simplifying the syntax tree of an expression (but they do not necessarily make the expression smaller). They usually involve a pair of introduction-

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M \equiv M : A}\ \text{(refl)} \qquad \frac{\Gamma \vdash C \div A}{\Gamma \vdash C \equiv C \div A}\ \text{(refl')}$$

$$\frac{\Gamma \vdash M \equiv N : A}{\Gamma \vdash N \equiv M : A}\ \text{(sym)} \qquad \frac{\Gamma \vdash C \equiv D \div A}{\Gamma \vdash D \equiv C \div A}\ \text{(sym')}$$

$$\frac{\Gamma \vdash M \equiv N : A \quad \Gamma \vdash N \equiv P : A}{\Gamma \vdash M \equiv P : A}\ \text{(trans)} \qquad \frac{\Gamma \vdash C \equiv D \div A \quad \Gamma \vdash D \equiv E \div A}{\Gamma \vdash C \equiv E \div A}\ \text{(trans')}$$

$$\frac{\Gamma \vdash M \longrightarrow_{\beta} N : A}{\Gamma \vdash M \equiv N : A}\ (\beta\text{-equiv}) \qquad \frac{\Gamma \vdash C \longrightarrow_{\beta'} D \div A}{\Gamma \vdash C \equiv D \div A}\ (\beta\text{'-equiv})$$

$$\frac{\Gamma \vdash M \longrightarrow_{\eta} N : A}{\Gamma \vdash M \equiv N : A}\ (\eta\text{-equiv}) \qquad \frac{\Gamma \vdash C \longrightarrow_{\eta'} D \div A}{\Gamma \vdash C \equiv D \div A}\ (\eta\text{'-equiv})$$

$$\frac{\Gamma \vdash M_1 \equiv M_2 : A \quad \Gamma \vdash N_1 \equiv N_2 : B}{\Gamma \vdash [M_1, N_1] \equiv [M_2, N_2] : A \times B}\ \text{(cong-pair)} \qquad \frac{\Gamma \vdash C_1 \equiv C_2 : A}{\Gamma \vdash \textbf{pure}\ C_1 \equiv \textbf{pure}\ C_2 \div A}\ \text{(cong'-pure)}$$

$$\frac{\Gamma \vdash M \equiv N : A\ \mathsf{always}}{\Gamma \vdash \mathsf{extract}\ M \equiv \mathsf{extract}\ N : A\ \mathsf{now}}\ \text{(cong-extract)} \qquad \frac{\begin{array}{c}\Gamma \vdash E_1 \equiv E_2 : \mathsf{Event}\ A\ \mathsf{now} \\ \Gamma^{\mathsf{s}}, A\ \mathsf{now} \vdash C \div B\ \mathsf{now}\end{array}}{\begin{array}{l}\Gamma \vdash \textbf{let evt}\ x = E_1\ \textbf{in}\ C \\ \quad \equiv \textbf{let evt}\ x = E_2\ \textbf{in}\ C \div B\ \mathsf{now}\end{array}}\ \text{(cong'-letEvt)}$$

Figure 3.5 – Term equality rules.

elimination rules which can be "cancelled out" in some way. The classic example is the β-reduction of the λ-calculus:

$$(\lambda x.\, M)\, N \xrightarrow{\lambda}_{\beta} [M/x]N$$

Note that this rule (and the rules for sums) involves substitution, so the resulting term is not equal to any of the ones that appear in the original expression. This is not the case for other rules, such as the reduction of products:

$$\mathsf{fst}\ [M, N] \xrightarrow{\mathsf{fst}}_{\beta} M \qquad \mathsf{snd}\ [M, N] \xrightarrow{\mathsf{snd}}_{\beta} N$$

Our language has two additional constructor-destructor term pairs for stable types and events. The reduction rules for these follow the ones proposed by Pfenning and Davies (2001). First, constructing a stable type and then binding it in a term or computation can be replaced by substitution:

$$\text{let stable}\, x = (\text{stable}\, M)\, \text{in}\, N \xrightarrow{\text{sta}}_{\beta} [M/x]N \qquad \textbf{let stable}\, x = (\text{stable}\, M)\, \textbf{in}\, C \xrightarrow{\text{sta}'}_{\beta'} [M/'x]C$$

As events can only be destructed in computations, they have only one reduction rule:

$$\textbf{let evt}\ x = (\text{event}\ C)\ \textbf{in}\ D \xrightarrow{\text{evt}}_{\beta'} \langle C/x \rangle D$$

At the moment, the only reduction rule we can add for select is handling two pure events. For more complex interactions, we need to add a new term for explicit waiting or a more refined notion of substitution – our attempts show that neither of these is straightforward. This will be one of the main directions for future work.

$$\textbf{select}\ (\text{event}\ (\textbf{pure}\ M_1))\ \textbf{as}\ x \mapsto C_1 \parallel (\text{event}\ (\textbf{pure}\ M_2))\ \textbf{as}\ y \mapsto C_2 \parallel \textbf{both as}\ x, y \mapsto C_3$$
$$\xrightarrow{\text{select}_3}_{\beta'} [M_1/'x][M_2/'y]C_3$$

These temporal rules use three different types of substitution to account for the two kinds of expressions we use: term-into-term ($[M/x]$), term-into-computation ($[M/'x]$) and computation-into-computation ($\langle C/x \rangle$). Substituting computations into terms would not be possible, as it would allow us to access future events in the present.

3.2.2 Expansion rules

Expansion rules express a form of "extensional equality": two terms are the same if they have the same behaviour. Given a term of some type, we can construct a compound expression of the same type by applying its introduction and elimination rules. Common examples are η-expansion for functions and pairs:

$$M : A \to B \xrightarrow{\lambda}_{\eta} \lambda x.\, M\, x \qquad (x \notin fv(M))$$
$$M : A \times B \xrightarrow{\text{pair}}_{\eta} [\text{fst}\ M, \text{snd}\ M]$$

The η-rules for stable types bind the term to x, then returns the term stable x:

$$M : \text{Stable}\ A \xrightarrow{\text{sta}}_{\eta} \text{let stable}\ x = M\ \text{in}\ (\text{stable}\ x)$$
$$\textbf{pure}\ M \div \text{Stable}\ A \xrightarrow{\text{sta}'}_{\eta'} \textbf{let stable}\ x = M\ \textbf{in}\ (\textbf{pure}\ (\text{stable}\ x))$$

The rule for events is unusual due to the two representations of events as terms of type Event A and computations of type A.

$$M : \text{Event}\ A \xrightarrow{\text{evt}}_{\eta} \text{event}\ (\textbf{let evt}\ x = M\ \textbf{in}\ (\textbf{pure}\ x))$$

3.2.3 Equality relation

To express the reduction and expansion rules in an equational manner, we introduce the judgement $\Gamma \vdash M \equiv N : A$ and $\Gamma \vdash C \equiv D \div A$ for the equality of terms and computations. We want term equality to be an equivalence relation and congruence, and to be closed under

β-reduction and η-expansion. We show the relevant inference rules in Fig. 3.5. $\Gamma \vdash M \longrightarrow_x N : A$ is a shorthand for the two typing judgements for M and N and the assumption that $M \longrightarrow_x N$. We show only two congruence rules as an example, the others are all standard.

3.2.4 Agda implementation

As most inductively defined relations, term equality in Agda is expressed as a datatype with one constructor for each inference rule. It has the following signature:

```
data Eq (Γ : Context) : (A : Judgement) -> Γ ⊢ A -> Γ ⊢ A -> Set
```

For clarity, we define custom syntax for these judgements using Agda's syntax declaration:

```
syntax Eq Γ A M N = Γ ⊢ M ≡ N :: A
```

The equality judgements can now be expressed in a very natural way, for example:

```
data Eq Γ where
  sym    : ∀{A}{M N : Γ ⊢ A} ->      Γ ⊢ M ≡ N :: A
                                   ---------------
                                ->      Γ ⊢ N ≡ M :: A

  β-lam : ∀{A B}   ->  (N : Γ, A now ⊢ B now)   (M : Γ ⊢ A now)
                       -------------------------------------------
                   ->     Γ ⊢ (lam N) $ M ≡ [M /] N :: B now

  η-evt : ∀{A}     ->            (M : Γ ⊢ Event A now)
                          --------------------------------------------
                   ->      Γ ⊢ M ≡ event (letEvt M In pure (var top))
                                                   :: Event A now
  cong-stable : ∀{A}{M N : Γˢ ⊢ A now}
                   ->              Γˢ ⊢ M ≡ N :: A now
                           -----------------------------------
                   ->      Γ ⊢ stable M ≡ stable N :: A always
```

The syntax and rules are similar for computation equality.

In this chapter we described the syntax of our language and showcased the main aspects of the Agda implementation. Next, we turn to category theory and give a sound categorical semantics for the language.

CHAPTER 4

Categorical foundations

This chapter introduces the background required to interpret modal logic in a categorical setting, and sets the foundations for our discussion of denotational semantics in the next chapter. We begin with a brief literature review and describe how the ideas of modal and temporal logic can be translated into the language of category theory. Then, we present the concrete category that we will be working in. A summary of the relevant parts of category theory, and the terminology and notation we use in the rest of this dissertation can be found in Appendix A.

4.1 Modalities in category theory

4.1.1 The Curry–Howard–Lambek correspondence for modal logic

The *Curry–Howard correspondence* (Howard, 1980) is a well-known connection between constructive logic and type theory. It states that a proposition in logic can be interpreted as the type of an expression (in a typed λ-calculus), and the expression is a constructive proof term for this proposition. We saw this in the last chapter: our language is essentially a system of proof terms for temporal logic.

In the 1970s, Joachim Lambek (1980) extended this connection to category theory, discovering that the simply typed lambda calculus corresponds to Cartesian closed categories (Lawvere, 1964), i.e. it can be interpreted in any category which has finite products and exponentials. This observation advanced the fields of *categorical semantics* and *categorical logic*, which aim to generalise the study of formal languages and logics by finding exactly the properties that a category needs to be a model of the formal system.

Given these deep parallels between the three subjects (which Harper (2011) calls *computational trinitarianism*), we might wonder if extensions to logic (e.g. modalities) translate to useful properties of type systems and categories. A full account might be elusive, purely because modal logic represents many different systems and has no universal definition. However, considering specific modal logics such as temporal logic seems to be more fruitful, and this project aims to make a contribution to this research.

As discussed in Section 2.3, there is a strong connection between (linear) temporal logic and type systems for functional reactive programming. Jeffrey (2012, 2014) formalises this connection in Agda as a dependent type system for future and past time LTL. There has also been a long line of work on connecting (constructive) temporal logics such as S4 (Prawitz, 1965) with category theory, which – curiously enough – was motivated by the practical need to model computations:

- Moggi's (1991) influential paper introduced the *computational lambda calculus* (CLC), and its denotational semantics in a CCC with a strong monad to model computations.
- Fairtlough and Mendler (1994) develop a intuitionistic modal logic called *propositional lax logic* with a single modality that obeys both possibility- and necessity-like axioms. They state that its categorical semantics correspond to the CLC, but mainly examine the Kripke semantics.
- Kobayashi (1997) adapts Moggi's semantics to a model of S4 modal logic by introducing the more general notion of an U-strong monad.
- Benton, Bierman, and de Paiva (1998) establish a logic based on CLC called CL-logic, with a modality for possibility that can be modelled by a strong monad.
- Bierman and de Paiva (2000) develop an intuitionistic version of the S4 along with a categorical model based on Kobayashi's modalities.
- Alechina et al. (2001) take a similar route, comparing the algebraic, Kripke and categorical semantics of constructive S4 logic and propositional lax logic.
- Bellin, de Paiva, and Ritter (2001) formulate an intuitionistic version of the K modal logic with a non-monadic categorical semantics.
- Jeltsch (2012) explores a common categorical semantics for linear temporal logic and functional reactive programming, introducing the notion of a *temporal category*. In further research (Jeltsch, 2013, 2014a,b) he extends temporal categories to *abstract process categories* which can model FRP processes and the $\mathcal{U}$-modality.
- de Paiva and Ritter (2016) extend a $\Box$-only intuitionistic S4 to a dependently typed modal logic and interpret it in a fibred categorical semantics.
- de Paiva and Eades (2017) describe a past- and future-time tense logic with a double intertwined adjoint categorical model of modalities.

In the rest of this section we will look at some of these developments in turn, giving a foundation to the concrete category we consider in the next section.

4.1.2 Monads as modalities?

The fact that we should suspect a connection between monads and modalities in a constructive modal logic becomes quite apparent when we look at the Hilbert-style axiomatisation of S4[1]:

$$\begin{aligned}
&\mathbf{K}\text{-}\Box : \Box(A \to B) \to (\Box A \to \Box B) &\qquad& \mathbf{K}\text{-}\Diamond : \Box(A \to B) \to (\Diamond A \to \Diamond B)\\
&\mathbf{T}\text{-}\Box : \Box A \to A && \mathbf{T}\text{-}\Diamond : A \to \Diamond A\\
&\mathbf{4}\text{-}\Box : \Box A \to \Box\Box A && \mathbf{4}\text{-}\Diamond : \Diamond\Diamond A \to \Diamond A
\end{aligned}$$

[1]In general, the possibility and necessity modalities in a non-classical modal logic are not interderivable.

The **K** axioms represent some sort of distributivity over implication, while the **T** and **4** axioms look suspiciously like the comonad and monad natural transformations. As we will see, these similarities are not accidental.

One of the first attempts to establish a connection between computer science and category theory was Moggi's seminal work on the computational lambda calculus (Moggi, 1991). He proposed modelling impure computations $f\colon A \rightsquigarrow B$ as Kleisli morphisms $f\colon A \to TB$ for a monad T. The monad operations η and μ, and in particular the derived extension operator $\_^{\star}\colon (A \Rightarrow TB) \to (TA \Rightarrow TB)$ can express the sequencing of computations. Monads have become a staple concept of practical functional programming ever since Wadler (1995) introduced them to Haskell (Hudak, Hughes, et al., 2007).

While the Curry–Howard correspondence usually translates logical concepts to type theory, Benton, Bierman, and de Paiva (1998) went the other direction and developed a logical analogue of Moggi's CLC. The resulting *CL-logic* is an intuitionistic logic with a "curious" possibility-like modality $\Diamond$ modelled by the same categorical structure as the computation type constructor T. Around the same time, Fairtlough and Mendler (1994) introduced the *propositional lax logic*, an intuitionistic logic with a "peculiar" modality $\bigcirc$ – though their motivation was different (hardware verification), they arrived to the same CL-logic as Benton, Bierman, and de Paiva.

The denotational semantics of the CLC, CLL and PLL requires T to be not just a monad, but a *strong monad*, as otherwise the modality elimination term would not have a suitable denotation (Moggi, 1991, Remark 3.1). Strong monads were introduced by Kock (1972) as a specialisation of *strong endofunctors* in a Cartesian (or monoidal) category. In a closed category with a strong endofunctor T, the existence of the natural transformation $(A \Rightarrow B) \to (TA \Rightarrow TB)$ is equivalent to the existence of a *tensorial strength* for T.

Definition. A *strong monad T* on a Cartesian category $\mathbb{C}$ is a monad together with a natural transformation st (tensorial strength) with components

$$\mathrm{st}_{A,B}\colon A \times TB \to T(A \times B)$$

such that some unit, associativity and strong naturality conditions hold.

We will not detail the coherence conditions here – as it turns out, strong monads are precisely what we do *not* want for a model of temporal logic.

The reason why Benton, Bierman, and de Paiva, and Fairtlough and Mendler consider their modalities[2] odd is that they do not conform to any commonly used modal logic: the modalities $\Diamond$ and $\bigcirc$ have the characteristics of necessity and possibility, i.e. axioms that are associated with both modalities. While for the intended purposes – computations and hardware verification – these make sense, they are not suitable for temporal reasoning.

Kobayashi (1997) points out that since these logics are modelled by CCCs with strong monads, $A \wedge \Diamond B \implies \Diamond(A \wedge B)$ should be provable in them. In a temporal logic such as S4, this is certainly not the case: if we know that it's raining now and it will eventually stop raining, we cannot conclude that eventually it will be both raining and not raining. What this means is that the usual interpretation of computational calculi and logics as a category with strong monads is not sound for temporal logic.

[2]Benton, Bierman, and de Paiva point out that a similar constructive modal logic was considered even by Curry (1952), but he found it too unusual to explore further.

4.1.3 More general notion of strength

To resolve this, Kobayashi introduces a more general notion of strength called U-strength, which corresponds to the logical expression $\Box A \wedge \Diamond B \implies \Diamond(\Box A \wedge B)$. Fortunately, this is derivable in S4 in an intuitive way: the truth of A will persist for any eventual future. Kobayashi requires the new modality $\Box$ to be interpreted as a *Cartesian comonad*, which also follows from the axioms of $\Box$ in S4. We now give the full definition of both terms; the exact coherence laws will be given later, specialised to our modalities.

Definition. A *Cartesian endofunctor F* on a Cartesian category $\mathbb{C}$ is a functor which preserves finite products, that is, there exists:

- a morphism $\mathrm{u} \colon \top \to F\top$
- a natural transformation with components $\mathrm{m}_{A,B} \colon FA \times FB \Rightarrow F(A \times B)$

such that F respects the unit and associativity laws of $\mathbb{C}$.

Definition. A *Cartesian comonad U* on a Cartesian category $\mathbb{C}$ is a comonad for which there exist maps u and m to make $(U, \mathrm{u}, \mathrm{m})$ a Cartesian functor, and the comonadic natural transformations interact with the Cartesian structure in a sound way.

Definition. Let $\mathbb{C}$ be a Cartesian category with a Cartesian comonad U. For an endofunctor F on $\mathbb{C}$, we define the *U-tensorial strength* as the natural transformation st^U with components

$$\mathrm{st}^U_{A,B} \colon UA \times FB \to F(UA \times B)$$

which respects the Cartesian structure of $\mathbb{C}$.

Definition. Let $\mathbb{C}$ be a Cartesian category with a Cartesian comonad U. A *U-strong monad T* on $\mathbb{C}$ is a monad with a U-tensorial strength st^U consistent with the monadic structure of T.

Kobayashi then shows that constructive S4 modal logic has a sound interpretation in a CCC with a Cartesian comonad $\Box$ and a $\Box$-strong monad $\Diamond$. On Figs. 4.1 to 4.4 we give the coherence laws in such a model, specifying the interactions between the Cartesian, monadic and comonadic structure.

4.1.4 Temporal categories

Treating $\Box$ as a Cartesian comonad and $\Diamond$ as a $\Box$-strong monad has been the primary way of modelling S4 and temporal logic in subsequent research. Alechina et al. (2001) compare the algebraic, Kripke, and categorical semantics of PLL and CS4, using Moggi-style strong monads and Kobayashi-style $\Box$-strong monads respectively. Bierman and de Paiva (2000) also describe an intuitionistic S4 modal logic using the same formulation. Jeltsch (2012), in developing a correspondence between FRP and linear temporal logic, also arrives to the same model, albeit extending it to ideal monads (Aczel et al., 2003) to account for future-only modalities. His work also has other relevant aspects which are used in our denotational semantics.

Figure 4.1 – Cartesian endofunctor laws

$$\begin{tikzcd}
\Box A \times \top \arrow[r, "\rho"] \arrow[d, "\mathrm{id}_{\Box A}\times \mathrm{u}"'] & \Box A \\
\Box A \times \Box \top \arrow[r, "\mathrm{m}_{A,\top}"'] & \Box(A \times \top) \arrow[u, "\Box\rho"']
\end{tikzcd}$$

(a) Right unit

$$\begin{tikzcd}
\Box \top \times A \arrow[r, "\lambda"] \arrow[d, "\mathrm{u}\times \mathrm{id}_{\Box A}"'] & \Box A \\
\Box \top \times \Box A \arrow[r, "\mathrm{m}_{\top,A}"'] & \Box(\top \times A) \arrow[u, "\Box\lambda"']
\end{tikzcd}$$

(b) Left unit

$$\begin{tikzcd}
(\Box A \times \Box B) \times \Box C \arrow[r, "\alpha"] \arrow[d, "\mathrm{m}_{A,B}\times \mathrm{id}_{\Box C}"'] & \Box A \times (\Box B \times \Box C) \arrow[d, "\mathrm{id}_{\Box A}\times \mathrm{m}_{B,C}"] \\
\Box(A \times B) \times \Box C \arrow[d, "\mathrm{m}_{A\times B,C}"'] & \Box A \times \Box(B \times C) \arrow[d, "\mathrm{m}_{A,B\times C}"] \\
\Box((A \times B) \times C) \arrow[r, "\Box\alpha"'] & \Box(A \times (B \times C))
\end{tikzcd}$$

(c) Associativity

Figure 4.2 – Cartesian comonad laws

$$\begin{tikzcd}
\top \arrow[d, "\varepsilon_\top"'] \arrow[dr, equal, "\mathrm{id}_\top"] & \\
\Box \top \arrow[r, "\mathrm{u}"'] & \top
\end{tikzcd}$$

(a) Unit map and extract

$$\begin{tikzcd}
\top \arrow[r, "\mathrm{u}"] \arrow[d, "\mathrm{u}"'] & \Box \top \arrow[d, "\delta_\top"] \\
\Box \top \arrow[r, "\Box\mathrm{u}"'] & \Box\Box \top
\end{tikzcd}$$

(b) Unit map and duplicate

$$\begin{tikzcd}
\Box A \times \Box B \arrow[rr, "\mathrm{m}_{A,B}"] \arrow[dr, "\varepsilon_A\times\varepsilon_B"'] & & \Box(A \times B) \arrow[dl, "\varepsilon_{A\times B}"] \\
& A \times B &
\end{tikzcd}$$

(c) Multiplication map and unit

$$\begin{tikzcd}
\Box A \times \Box B \arrow[rr, "\delta_A\times\delta_B"] \arrow[d, "\mathrm{m}_{\Box A,\Box B}"'] & & \Box\Box A \times \Box\Box B \arrow[d, "\mathrm{m}_{\Box A,\Box B}"] \\
\Box(A \times B) \arrow[dr, "\delta_{A\times B}"'] & & \Box(\Box A \times \Box B) \arrow[dl, "\Box\mathrm{m}_{A,B}"] \\
& \Box\Box(A \times B) &
\end{tikzcd}$$

(d) Multiplication map and duplicate

Figure 4.3 – $\Box$-tensorial strength laws

$$\begin{tikzcd}
& \Box \top \times \Diamond A \arrow[dl, "\mathrm{st}^{\Box}_{\top,A}"'] \arrow[dr, "\lambda^{\Box}_{\Diamond A}"] & \\
\Diamond(\Box \top \times A) \arrow[rr, "\Diamond\lambda^{\Box}_{A}"'] & & \Diamond A
\end{tikzcd}$$

(a) Left unit and strength

$$\begin{tikzcd}
\Box(A \times B) \times \Diamond C \arrow[rr, "\alpha^{\Box}_{A,B,\Diamond C}"] \arrow[d, "\mathrm{st}^{\Box}_{A\times B,C}"'] & & \Box A \times (\Box B \times \Diamond C) \arrow[d, "\mathrm{id}_{\Box A}\times \mathrm{st}^{\Box}_{B,C}"] \\
\Diamond(\Box(A \times B) \times C) \arrow[dr, "\Diamond\alpha^{\Box}_{A,B,C}"'] & & \Box A \times \Diamond(\Box B \times C) \arrow[dl, "\mathrm{st}^{\Box}_{A,\Box B\times C}"] \\
& \Diamond(\Box A \times (\Box B \times C)) &
\end{tikzcd}$$

(b) Associativity and strength

Figure 4.4 – □-strong monad law

$$\begin{tikzcd}
\Box A \times B \arrow[d, "\mathrm{id}_{\Box A}\times\eta_B"'] \arrow[dr, "\eta_{\Box A\times B}"] & & \\
\Box A \times \Diamond B \arrow[r, "\mathrm{st}^{\Box}_{A,B}"] & \Diamond(\Box A\times B) & \\
\Box A \times \Diamond\Diamond B \arrow[u, "\mathrm{id}_{\Box A}\times\mu_B"] \arrow[r, "\mathrm{st}^{\Box}_{A,\Diamond B}"'] & \Diamond(\Box A\times\Diamond B) \arrow[r, "\Diamond\mathrm{st}^{\Box}_{A,B}"'] & \Diamond\Diamond(\Box A\times B) \arrow[ul, "\mu_{\Box A\times B}"']
\end{tikzcd}$$

Reactive types One common aspect of the work of Jeffrey (2012) and Jeltsch (2012) is their use of *time-indexed* or *reactive types*. Intuitively, the truth value of LTL propositions depends on time, so an LTL type system would have types with time-dependent inhabitance. Jeffrey expresses this as a dependent Agda type, while Jeltsch uses the product category $\mathbb{C}^T$ for a totally ordered set T representing times, and a BCCC $\mathbb{C}$. He defines *fan categories* as such a category with extra conditions allowing for bounded products and coproducts (which he uses in the definition of the □ and ◇ modalities). These fan categories are shown to have the required structure to model CS4.

Linearity As Jeltsch points out, categorical models for S4 logics do not capture the linearity of time: while logically there is nothing wrong with branching time (it is the basis of Computation Tree Logic (Clarke and Emerson, 1981), for example), it is not an intuitive model of FRP. To remedy this, he introduces *temporal categories* which, in addition to the usual CS4 structure, have a way to model the proposition

$$\Diamond A \wedge \Diamond B \implies \Diamond((A \wedge \Diamond B) \vee (\Diamond A \wedge B) \vee (A \wedge B))$$

In contrast to the proposition $\Diamond A \wedge \Diamond B \implies \Diamond(A \wedge B)$ (which is a theorem in CL and PLL, but not in S4), this states that if we know that A and B will eventually hold, then at some point one of three possibilities will be true: both A and B hold, or A holds and B holds later, or B holds and A holds later. In branching-time logics, this proposition would not necessarily be true: there may be different futures where the events happen at different times. In LTL, it expresses the fact that the time ordering relation is trichotomous, i.e. if $\Diamond A$ occurs at time t_A and $\Diamond B$ at time t_B, then either $t_A < t_B$, $t_A > t_B$ or $t_A = t_B$. This makes the indexing order a *strict total order*, which is a suitable abstraction for time.

Now we have the necessary theoretical background to define the concrete category that we will be working in.

4.2 The category of reactive types

In this section we introduce the concrete category that we will interpret our syntax in. The main features of this category – which we call **Reactive** – are:

- bicartesian closed structure;
- a naturally arising monoidal comonad as the $\Box$ modality;
- an efficiently implementable $\Box$-strong monad as the $\Diamond$ modality;
- weak temporal linearity.

4.2.1 The base category

We follow Jeffrey and Jeltsch in our definition of the base types. Given our Agda implementation, we will treat sets and types somewhat interchangeably: a type `Nat` is treated as the set $\mathbb{N}$, while `Set`, the type of all types, can be seen as the category **Set**.

Definition. A *reactive type A* is a map from natural numbers to sets, i.e. an element of $\mathbf{Set}^{\mathbb{N}}$. The kind of reactive types is denoted as $\tau := \mathbb{N} \to \mathbf{Set}$. Indexing (function application) will be written as A_n or $A|_n$.

The indexing set the set of (unordered) natural numbers: while this can only support discrete-time FRP, it is essential for our definition of the next and delay modalities.

Definition. The **Reactive** category is defined as:

- objects: reactive types $\tau := \mathbb{N} \to \mathbf{Set}$;
- arrows: dependent types $A \mathrel{\dot{\to}} B := \prod_{n \in \mathbb{N}}(A_n \to B_n)$
- identities: $\mathrm{id}_A|_n := \mathrm{id}_A$
- composition: $g \circ f|_n := g_n \circ f_n$

All category laws follow from the laws for **Set**, via pointwise application.

"Pointwise application" simply means that properties and laws hold at every time step, i.e. every index $n \in \mathbb{N}$. This lets us make use of a lot of the underlying structure of **Set** – in particular, that it is bicartesian closed.

Theorem. **Reactive** *is bicartesian closed.*

Proof. We define **Reactive** products, coproducts and exponentials by pointwise indexing. For example,

$$A \otimes B|_n := A_n \times B_n \qquad A \oplus B|_n := A_n \uplus B_n$$

Again, all laws transfer directly from the BCCC nature of **Set**. $\Box$

4.2.2 Modalities

The two modalities $\Box$ and $\Diamond$ are usually defined via two dual concepts, mirroring the "for all" and "exists" duality of their logical definitions. For example, Jeltsch (2012) uses infinite products and coproducts in a fan category; Jeffrey (2012) defines them with dependent product and sum types; Cave et al. (2014) employs dual coinductive-inductive types utilising the $\bigcirc$ modality and greatest-least fixed points; and Krishnaswami (2013) defines streams and events via a temporal recursive type on products and coproducts in the style of Nakano (2000). While elegant and easy to analyise theoretically, such representations make efficient implementations quite difficult. Going back to our discussion of push- and pull-based FRP, the issue is with event handling: we either have to keep polling (which would be required for an inductive coproduct-like definition of $\Diamond$), or resort to callbacks (making reasoning about the system really complicated).

Our main goal with the denotational semantics of the language was a definition of $\Diamond$ which would translate well to an efficient, push-based representation. Instead of saying that $\Diamond E$ represents an event that "can happen now, or on the next tick, or the next, etc.", we want to say the event "happens after some delay". Ultimately, we hope to show that such a representation translates into an implementation via callbacks or continuation-passing style, which would allow us to reason about events in an abstract way but still implement them efficiently.

Box modality

Our box modality deviates slightly from the usual definition of $\Box$ in temporal logic or LTL. There, $\Box A$ means "henceforth, A holds", with the Kripke semantics of "A holds in all worlds accessible from the current one". Both notions are indexed by the *current* time, making $\Box$ unsuitable for expressing globality over *all* times, past, present and future. We want the denotation of A always to express that the type A will always be inhabited, not just henceforth. That is, the box modality should correspond to some notion of constancy at all times.

We can arrive at a suitable definition by considering the relationship of normal types (**Set**) and reactive types (**Reactive**). The first models standard functional programming, while the second allows us to write time-dependent reactive programs. We should be able to turn every normal type T into a **Reactive** type, just by returning T at all time steps. However, we can also go from any reactive type $B : \tau$ to a normal type, by expressing B as a dependent function from time n to B_n. In fact, these two transformations can be defined as functors:

$$K \colon \mathbf{Set} \longrightarrow \mathbf{Reactive}$$
$$G \colon \mathbf{Reactive} \longrightarrow \mathbf{Set}$$

$$\begin{aligned}
K(T : \mathrm{Set})_n &= T \\
K(f : T \to S)_n(a : T) &= f(a) \\
G(A : \tau) &= \textstyle\prod_{n:\mathbb{N}} A_n \\
G(f : A \twoheadrightarrow B)(a : A) &= \lambda n : \mathbb{N}.\, f_n(a_n)
\end{aligned}$$

$$\begin{tikzcd}
\mathbf{Set} \arrow[rr, bend left, "K"] \arrow[rr, phantom, "\perp"] & & \mathbf{Reactive} \arrow[ll, bend left, "G"]
\end{tikzcd}$$

With these definitions, it is easy to prove that K and G are adjoint: $K \dashv G$ with unit $\eta_T \colon T \longrightarrow GKT = T \longrightarrow \prod_{n:\mathbb{N}} T$ and counit $\varepsilon_A \colon KGA \twoheadrightarrow A = (\lambda n : \mathbb{N}.\ \prod_{k:\mathbb{N}} A_k) \twoheadrightarrow A$. The unit is not particularly interesting: it turns a type into a constant dependent type. The counit, however, is more promising: its argument is the constant reactive type $\prod_{k:\mathbb{N}} A_k$. Reading the dependent product as universal quantification, this says "for all times k, A_k is inhabited" – this lines up exactly with what we want our box type to represent. Indeed, we can extract this modality as the comonad of the adjunction:

$$\begin{aligned}
&\Box = KG\colon \textbf{Reactive} \longrightarrow \textbf{Reactive}\\
&\quad \Box A|_n &&= \textstyle\prod_{k:\mathbb{N}} A_k\\
&\quad \varepsilon_A|_n\ (a : \Box A|_n) &&= a(n)\\
&\quad \delta_A|_n\ (a : \Box A|_n) &&= \lambda k : \mathbb{N}.\, a
\end{aligned}$$

It is also straightforward to show that this comonad is Cartesian, with unit map $u_n(t : \top_n) = \lambda k : \mathbb{N}.\, t$ and multiplication $\mathrm{m}_{A,B}|_n\ (a : \Box A|_n, b : \Box B|_n) = \lambda k : \mathbb{N}.\, (a_k, b_k)$. Thus, we arrive at one of our modalities: a Cartesian comonad $\Box$.

Diamond modality

To reiterate, we aim to define $\Diamond E$ to mean "event E happens with some delay". Hence, we need to state what is a "delay" and what is the interpretation of "some". To do this, we expand on the work of Krishnaswami (2013) and his use of the next-step modality as a primitive temporal operation.

Next-step modality Krishnaswami defines the $\bullet A$ modality for expressing that a value of type A will be available on the next time step (Krishnaswami and Benton (2011a,b) and Krishnaswami, Benton, and Hoffmann (2012) also use a similar modality, with different ways of representing time steps). However, the modality appears as an explicit type constructor in the syntax, with term-level introduction and elimination forms. Since we want to abstract away the individual time steps from the syntax, this modality only appears in our semantics. Specifically, it is an endofunctor on **Reactive**:

$$\begin{aligned}
&\bullet\colon \textbf{Reactive} \longrightarrow \textbf{Reactive}\\
&\bullet A|_0 = \top_0 \qquad && \bullet f|_0\ (a : \top_0) = a\\
&\bullet A|_{n+1} = A_n \qquad && \bullet f|_{n+1}\ (a : A_n) = f_n(a)
\end{aligned}$$

It represents "shifting" a reactive type by one step into the future. At time 0, we cannot produce any meaningful value other than $\top$, as the type only "starts" on the next time step. At a time $n+1$, we return the value the type would have had on the previous step. Note that this definition differs from the usual specification of the $\bigcirc$ modality in LTL, which would not work in our case: we would need to return a value now which is only available on the next step. This violates causality, as we would need to look into the future.

The $\bullet$ modality can be iterated any number of times to push a type further into the future. This defines a family of functors we call *k-step delays*, for any $k \in \mathbb{N}$:

$$\begin{aligned}
&\bullet^{-}\colon \mathbb{N} \longrightarrow \textbf{Reactive} \longrightarrow \textbf{Reactive}\\
&\bullet^0 A = A \qquad && \bullet^0 f = f\\
&\bullet^{k+1} A = \bullet(\bullet^k A) \qquad && \bullet^{k+1} f = \bullet(\bullet^k f)
\end{aligned}$$

Both $\bullet$ and $\bullet^k$ are Cartesian functors.

Diamond modality Now we are ready to give our definition of $\Diamond$, using the functors we introduced above. $\Diamond A$ represents a reactive type A with a k-step delay, for some k. Interpreting "for some k" as "exists a $k \in \mathbb{N}$" and translating into the language of dependent existential types, we get the following definition of the $\Diamond$ functor:

$$\Diamond : \textbf{Reactive} \to \textbf{Reactive}$$

$$\Diamond A|_n \ = \textstyle\sum_{k:\mathbb{N}} \ \bullet^k A|_n \qquad\qquad \Diamond f|_n \ (k : \mathbb{N}, a : \bullet^k A|_n) \ = (k, \bullet^k f|_n \ (a))$$

Recall that the dependent sum type $\sum_{x:A} B(x)$ is inhabited by a pair consisting of a value $x : A$ and value of type $B(x)$. In our case, the type $\Diamond A$ is inhabited by a number k (indicating the amount of delay) and a value of the k-delayed reactive type $\bullet^k A$. That is, instead of representing possibility by an infinite disjunction over all time steps (where we can only handle the event by continuous polling), it is encoded as a constructive existential type which is inhabited by the exact time step the event occurs at. This avoids having to loop or recursively check whether and when the event fires, because the occurrence time is available directly. However, the sigma type lets us hide that from the types, and only say that "the event happens at some future time" – exactly what our intuition would suggest for the meaning of "eventually".

Monadicity Our definition for $\Diamond$ is only useful for our purposes if it is a monad. This is not immediately obvious, and in fact, we were sceptical of this initially, suspecting that some changes are required to the definition to prove the monad laws. However, our "fears" turned out to be unfounded. First, we describe the monad operations, then sketch the proofs of the monad laws.

The unit $\eta_A \colon A \rightarrow\!\!\!\triangleright \Diamond A$ is straightforward: it turns a value of type A into an event that happens with no delay:

$$\eta_A|_n \ (a : A_n) = (0, a)$$

Multiplication $\mu_A \colon \Diamond\Diamond A \rightarrow\!\!\!\triangleright \Diamond A$ is trickier. Logically, a type delayed by k and l is the same as one delayed by $k + l$, but we also need to take into account the current time, and whether one of the events has already happened. Suppose we are at time n and the argument of μ is the pair $(k, a : \bullet^k \Diamond A|_n)$. We need to compare whether the current time is before or after the occurrence of the outer event. If before, then the inner event a is delayed by more than the current time, so its type $\bullet^k \Diamond A|_n$ cannot be anything other than $\top$. If we are after the occurrence of the outer event (i.e. $k \le n$), then the type $\bullet^k \Diamond A|_n$ is equal to the type $\Diamond A|_{n-k}$, and unwrapping a gives the pair $(l, b : \bullet^l A|_{n-k})$. Now, the type of b can be rewritten to $\bullet^{k+l} A|_n$, and hence $(k + l, b : \bullet^{k+l} A|_n)$ is the return value of the correct type $\Diamond A|_n$.

$$\mu_A|_n \ (k, a : \bullet^k \Diamond A|_n)) = \begin{cases} \textbf{let}\ (l, b : \bullet^l A|_{n-k}) = (a \parallel \Diamond A|_{n-k})\ \textbf{in}\ (k + l, b \parallel \ \bullet^{k+l}\ A|_n), & \text{if } k \le n \\ \textbf{let}\ * = (a \parallel \top_0)\ \textbf{in}\ (k, * \parallel \ \bullet^k\ A|_n), & \text{if } k > n \end{cases}$$

Here, $(a \parallel B)$ marks a coercion of a value $a : A$ to $a : B$ assuming $A \equiv B$. All equality proofs are instances of the following lemma:

Lemma (Cancellation of delays). *For any $n, k, l \in \mathbb{N}$ and type $A : \tau$, we have*

$$\bullet^{l+k} A|_{l+n} = \bullet^{k} A|_n$$

Proof. By induction on $k \in \mathbb{N}$. □

The lemma says that extra delay and extra waiting can be cancelled out. In particular, if $k = 0$, then we have a shifted type without a delay: $\bullet^{l} A|_{l+n} = A_n$. If $n = 0$, we have $\bullet^{l+k} A|_{l} = \bullet^{k} A|_0$: this is equal either to A_0 (if $k = 0$) or $\top_0$ (if $k > 0$). A corollary of the latter is that the value $*$ (the inhabitant of $\top_n$) can have any delayed type, as long as the delay is longer than the elapsed time:

Corollary. *For any $n, k \in \mathbb{N}$ and type $A : \tau$, we have*

$$\top_n = \bullet^{n+(k+1)} A|_n$$

This raises a question: in the $k > n$ case of the definition of μ, we return $(k, * \parallel \bullet^{k} A|_n)$, but couldn't we just as well return $(k+1, * \parallel \bullet^{k+1} A|_n)$ or $(k+m, * \parallel \bullet^{k+m} A|_n)$ for any m? We anticipated this to be a source of ambiguity when proving the monad laws, and prepared a "plan B" of quotienting the definition of $\diamond$ by an equivalence relation which would identify such terms. However, there was no need for this, as the Agda formalisation only accepted the definition given above (in fact, it automatically deduced that the delay must be k). It is not entirely clear to us why that is the case, but it certainly saved us a lot of work and made the model a lot simpler than it could have been.

Theorem. *$\diamond$, together with η and μ satisfies the monad laws.*

Proof. We give a sketch of the proof, specifying the main lemmas that it uses. For simplicity, we extract the first case of the definition of μ into a general operation $(\triangleright^{k}_{n})_A : \diamond A|_n \rightarrow \diamond A|_{k+n}$ for shifting an event at time n to one at time $k + n$:

$$(\triangleright^{k}_{n})_A(a : \diamond A|_n) = \textbf{let } (l, b : \bullet^{l} A|_n) = a \textbf{ in } (k + l, b \parallel \bullet^{k+l} A|_{k+n})$$

Thus, the definition of μ becomes

$$\mu_A|_n\ (k, a : \bullet^{k} \diamond A|_n)) = \begin{cases} (\triangleright^{k}_{n-k})_A(a \parallel \diamond A|_{n-k}), & \text{if } k \leq n \\ (k, * \parallel \bullet^{k} A|_n), & \text{if } k > n \end{cases}$$

In fact, we can show that $\triangleright^{k}_{n} : (\diamond - |_n) \Longrightarrow (\diamond - |_{n+k})$ is a natural transformation between the functors $\diamond - |_n, \diamond - |_{n+k} :$ **Reactive** $\rightarrow$ **Set**. Below is the naturality condition which is used to prove the naturality of μ:

$$\begin{tikzcd} A \arrow[d, "f"] & \diamond A|_n \arrow[r, "(\triangleright^{k}_{n})_A"] \arrow[d, "\diamond f|_n"'] & \diamond A|_{k+n} \arrow[d, "\diamond f|_{k+n}"] \\ B & \diamond B|_n \arrow[r, "(\triangleright^{k}_{n})_B"'] & \diamond B|_{k+n} \end{tikzcd}$$

We also have the following lemma, which replaces two successive event shifts with one.

Lemma (Composition of event shifting). *For all $n, k, l \in \mathbb{N}$,*

$$\triangleright^{l}_{k+n} \circ \triangleright^{k}_{n} = \triangleright^{l+k}_{n}$$

This is used in the proof of the following interchange theorem between μ and $\triangleright^k_n$:

Lemma (Interchange of shifting and multiplication). *The order of shifting and multiplication can be swapped.*

$$\mu_A|_{n+k} \circ (\triangleright^k_n)_{\Diamond A} = (\triangleright^k_n)_A \circ \mu_A|_n$$

$$\begin{tikzcd}
\Diamond\Diamond A|_n \arrow[r, "(\triangleright^k_n)_{\Diamond A}"] \arrow[d, "\mu_A|_n"'] & \Diamond\Diamond A|_{k+n} \arrow[d, "\mu_A|_{k+n}"] \\
\Diamond A|_n \arrow[r, "(\triangleright^k_n)_A"'] & \Diamond A|_{k+n}
\end{tikzcd}$$

Now we are ready to prove the monad laws. The Agda formalisation is quite complicated due to the need for explicit coercions of terms and congruences, but equational reasoning and heterogeneous equalities help out significantly. The full proof can be found in Appendix C; below we show one of the cases of the associativity law as an example.

Lemma (Associativity law). $\mu_A \circ \mu_{\Diamond A} = \mu_A \circ \Diamond \mu_A$

Proof. In **Reactive**, this equation expands to the following, for $k : \mathbb{N}$ and $a : \bullet^k \Diamond\Diamond A|_n$:

$$\mu_A|_n\ (\mu_{\Diamond A}|_n\ (k, a)) = \mu_A|_n\ (\Diamond \mu_A|_n\ (k, a))$$

We consider one of the cases:

Case $k \leq n$:

$$\begin{aligned}
\mu_A|_n\ (\mu_{\Diamond A}|_n\ (k, a)) &= \mu_A|_n\ ((\triangleright^k_{n-k})_{\Diamond A}(a \mid\mid \Diamond\Diamond A|_{n-k})) && \text{(def. of } \mu \text{ (case } k \leq n))\\
&= (\triangleright^k_{n-k})_A(\mu_A|_{n-k}\ (a \mid\mid \Diamond\Diamond A|_{n-k})) && \text{(interchange of } \mu \text{ and } \triangleright^k_{n-k})\\
&= (\triangleright^k_{n-k})_A(\bullet^k \mu_A|_n\ (a) \mid\mid \Diamond A|_{n-k})) && \text{(delay lemma)}\\
&= \mu_A|_n\ (k, \bullet^k \mu_A|_n\ (a)) && \text{(def. of } \triangleright^k_{n-k})\\
&= \mu_A|_n\ (\Diamond \mu_A|_n\ (k, a)) && (\Diamond \text{ map on morphisms})
\end{aligned}$$

■

By proving the other case and the unit laws, we conclude that $\Diamond$ is a monad. □

Further properties

Relationship of the modalities Above we described four modality-like operators: $\Diamond$, $\Box$, $\bullet$ and $\bullet^k$. Though we do not have the duality of $\Diamond$ and $\Box$ taken as an axiom in classical modal logic, we can consider the logical implications between them. As the modalities are functors, these implications can be translated into natural transformations.

- If A always holds, it will hold after a delay by k.
- If A holds after a delay by k, it holds after an unknown delay.
- A holds after a delay by 1 if and only if it holds on the next time step.

$$\Box \xRightarrow{\Box\text{-}\bullet^k} \bullet^k \xRightarrow{\bullet^k\text{-}\Diamond} \Diamond \qquad \bullet^1 \xRightarrow{\cong} \bullet$$

$\Box$-tensorial strength Our definition of $\Diamond$ is not a strong monad: we cannot define a function $A \otimes (\Diamond B) \mathrel{\multimap} \Diamond(A \otimes B)$ as we cannot guarantee that A is inhabited when the event $\Diamond B$ fires. This is exactly the difficulty that Kobayashi anticipated and which can be resolved by only requiring $\Diamond$ to be a *$\Box$-strong monad*. Establishing this amounts to defining a natural transformation $\mathrm{st}^\Box$ and proving that it is consistent with the Cartesian comonad structure of $\Box$.

$$\mathrm{st}^\Box_{A,B} \colon \Box A \otimes \Diamond B \mathrel{\multimap} \Diamond(\Box A \otimes B)$$
$$\mathrm{st}^\Box_{A,B}|_n\ (a, (k, b)) = (k, \mathrm{m}^{\bullet^k}_{\Box A,B}|_n\ (\Box\text{-}\bullet^k_{\Box A}|_n\ (\delta_A|_n\ a), b))$$

Given a signal $a : \Box A$ and event $(k, b) : \Diamond B$, we construct a new event with the same occurrence time k. We get its value by applying the multiplication of the Cartesian functor $\bullet^k$ to $b : \bullet^k A$ and $\Box\text{-}\bullet^k_{\Box A}|_n\ (\delta_A|_k\ a) : \bullet^k \Box A$ which uses the $\Box\text{-}\bullet^k$ natural transformation we presented above to delay $\Box A$ by k extra steps. To prove that this is the right definition of $\Box$-tensorial strength, we need to show that the diagrams in Figs. 4.3 and 4.4 commute. The proofs are quite lengthy but mostly rely on naturality conditions and Cartesian functor laws.

Linearity The temporal linearity property would allow us to interpret the select operation in our semantics. In Jeltsch's definition, linearity means the existence of a morphism

$$\Diamond A \otimes \Diamond B \to \Diamond((A \otimes \Diamond B) \oplus (\Diamond A \otimes B) \oplus (A \otimes B))$$

for all A and B. Jeltsch recognises this as being equivalent to requiring that

$$A \circledast B := (A \otimes \Diamond B) \oplus (\Diamond A \otimes B) \oplus (A \otimes B)$$

is the product of A and B in the Kleisli category of $\Diamond$, with the linear-time product of the morphisms $f \colon C \to \Diamond A$ and $g \colon C \to \Diamond B$ denoted as $\langle\!\langle f, g \rangle\!\rangle \colon C \to \Diamond(A \circledast B)$. We can define this operation in **Reactive** as follows: given two events (k, a) and $(k + l + 1, b)$, we combine their values with a function of type $\bullet^k A \otimes \bullet^{k+l+1} B \mathrel{\multimap} \bullet^k(A \otimes \Diamond B)$ (which uses the fact that $\bullet$ is Cartesian) and return a new event at time k with the value injected into the appropriate "slot" of the product. Specifying this operation is sufficient for our semantics, so we leave verifying that $\circledast$ is a product in **Reactive**$_\Diamond$ as future work.

CHAPTER 5

Semantics

This chapter combines the developments of the previous two and describes the denotational semantics of our language. First, we define the interpretation of types, terms and contexts in the **Reactive** category, and describe the approach for proving the soundness of substitution. Then, we prove the soundness of term equality and hence the soundness of our categorical semantics.

5.1 Semantics of types and terms

First, we present the interpretation of the syntactic constructs in the language. Most of the developments come from the standard categorical semantics of the λ-calculus (e.g. Crole, 1993), augmented with the temporal operators and qualifiers which translate naturally into our **Reactive** category.

5.1.1 Types and contexts

The "motto" of categorical semantics is *types are objects, terms are morphisms*. Our basic types are readily interpreted as objects in a BCCC, and the additional event and stable types map to the temporal modalities we described in the previous section. The temporal qualifiers affect the modality of the underlying type: persistent types are interpreted as types under the $\Box$ modality, expressing that they are always available. Finally, contexts are interpreted as the product of the denotation of judgements in the context.

$$\begin{aligned}
[\![\ \mathsf{Unit}\ ]\!]_{\mathrm{t}} &= \top \\
[\![\ A \times B\ ]\!]_{\mathrm{t}} &= [\![A]\!]_{\mathrm{t}} \otimes [\![B]\!]_{\mathrm{t}} \\
[\![\ A + B\ ]\!]_{\mathrm{t}} &= [\![A]\!]_{\mathrm{t}} \oplus [\![B]\!]_{\mathrm{t}} \\
[\![\ A \to B\ ]\!]_{\mathrm{t}} &= [\![A]\!]_{\mathrm{t}} \Rightarrow [\![B]\!]_{\mathrm{t}} \\
[\![\mathsf{Stable}\ A]\!]_{\mathrm{t}} &= \Box[\![A]\!]_{\mathrm{t}} \\
[\![\ \mathsf{Event}\ A\ ]\!]_{\mathrm{t}} &= \Diamond[\![A]\!]_{\mathrm{t}}
\end{aligned}
\qquad
\begin{aligned}
[\![\ A\ \mathsf{now}\ ]\!]_{\mathrm{j}} &= [\![A]\!]_{\mathrm{t}} \\
[\![\ A\ \mathsf{always}\ ]\!]_{\mathrm{j}} &= \Box[\![A]\!]_{\mathrm{t}} \\
\\
[\![\ \cdot\ ]\!]_{\mathrm{x}} &= \top \\
[\![\Gamma, A]\!]_{\mathrm{x}} &= [\![\Gamma]\!]_{\mathrm{x}} \otimes [\![A]\!]_{\mathrm{j}}
\end{aligned}$$

Context stabilisation was an important operation in our syntax – what would it correspond to in the semantics? It could be a natural transformation with components $[\![^{\mathrm{s}}]\!]_{\Gamma} : [\![\Gamma]\!]_{\mathrm{x}} \rightarrow [\![\Gamma^{\mathrm{s}}]\!]_{\mathrm{x}}$ between the functors $[\![-]\!]_{\mathrm{x}}$ and $[\![-^{\mathrm{s}}]\!]_{\mathrm{x}}$ from the category of contexts to **Reactive**. However, any order-preserving embedding $\Delta \subseteq \Gamma$ induces a morphism $[\![\Gamma]\!]_{\mathrm{x}} \rightarrow [\![\Delta]\!]_{\mathrm{x}}$, so $\Gamma^{\mathrm{s}} \subseteq \Gamma$ is just an instance of that standard construction. Instead, we make use of the fact that all types in $\Gamma^{\,s}$ are persistent and therefore interpreted as boxed reactive types. Recall that $\Box$ is monoidal, so a product of boxed types can be transformed into a boxed product – moreover, as $\Box$ is a comonad, we can duplicate the boxes first and then factor one out:

$$x : A\ \text{always}, y : B\ \text{now}, z : C\ \text{always} \xrightarrow{\llbracket \Gamma \rrbracket_{\mathrm{x}}} \Box A \otimes \Box C \xrightarrow{\delta} \Box\Box A \otimes \Box\Box C \xrightarrow{m} \Box(\Box A \otimes \Box C)$$

Thus, we define context stabilisation as a natural transformation between $\llbracket - \rrbracket_{\mathrm{x}}$ and $\Box\llbracket -^{\,s} \rrbracket_{\mathrm{x}}$:

$$\llbracket^{s}\rrbracket\Box : \llbracket - \rrbracket_{\mathrm{x}} \Longrightarrow \Box\llbracket -^{\,s} \rrbracket_{\mathrm{x}}$$

$$\begin{aligned}
\llbracket^{s}\rrbracket\Box_{(\cdot)} &= \mathrm{u} \\
\llbracket^{s}\rrbracket\Box_{(\Gamma, A\ \text{now})} &= \llbracket^{s}\rrbracket\Box_{\Gamma} \circ \pi_1 \\
\llbracket^{s}\rrbracket\Box_{(\Gamma, A\ \text{always})} &= \mathrm{m}_{\llbracket\Gamma\rrbracket_{\mathrm{x}}, \Box\llbracket A \rrbracket_{\mathrm{t}}} \circ (\llbracket^{s}\rrbracket\Box_{\Gamma} \times \delta_{\llbracket A \rrbracket_{\mathrm{t}}})
\end{aligned}$$

This definition can be used to derive the more general transformation $\llbracket^{s}\rrbracket$:

$$\llbracket^{s}\rrbracket = \varepsilon \circ \llbracket^{s}\rrbracket\Box$$

Next, we give the interpretations of terms.

5.1.2 Terms

In categorical semantics, terms of a language are interpreted as morphisms from the denotation of the context to the denotation of the type. Again, the non-reactive terms of our language are interpreted as BCCC morphisms in the standard way. Terms involving events and stable types use the additional monad and comonad properties of **Reactive**.

```
⟦_⟧m : ∀{Γ A} -> Γ ⊢ A -> (⟦Γ⟧x ⇾ ⟦A⟧j)
⟦     var top     ⟧m  =  π2
⟦   var (pop x)   ⟧m  =  ⟦var x⟧m ∘ π1
⟦      lam M      ⟧m  =  Λ⟦M⟧m
⟦      F $ M      ⟧m  =  ev ∘ ⟨⟦F⟧m, ⟦M⟧m⟩
⟦       unit      ⟧m  =  !
⟦      fst M      ⟧m  =  π1 ∘ ⟦M⟧m
⟦  extract {A} M  ⟧m  =  ε⟦A⟧t ∘ ⟦M⟧m
⟦  persist {Γ} M  ⟧m  =  □⟦M⟧m ∘ ⟦s⟧□Γ
⟦    stable M     ⟧m  =  ⟦M⟧m
⟦  letSta S In M  ⟧m  =  ⟦M⟧m ∘ ⟨id⟦Γ⟧x, ⟦S⟧m⟩
⟦     event C     ⟧m  =  ⟦C⟧c
```

As usual in the categorical semantics of the λ-calculus, variables, abstraction, application, products and sums are interpreted as the projection, injection, currying and evaluation morphisms of a BCCC. Applying `extract` to a persistent term amounts to extracting the value of the boxed denotation of the term. For `persist M` we use the $\llbracket^{s}\rrbracket\Box$ transformation to interpret `M` in the stabilised context. The term `stable M` is interpreted as $\llbracket \mathtt{M} \rrbracket_{\mathrm{m}}$, as both stable types and the always qualifier are translated into $\Box$. Stable binding first extends the context by the bound term, then interprets the body in this extended context. Finally, $[\![$`event C`$]\!]_{\mathrm{m}}$ calls the denotation of computations on `C`, shown below.

```
⟦_⟧c : ∀{Γ A} -> Γ ⊨ A -> (⟦Γ⟧x ⇸ ◇⟦A⟧j)
⟦    pure A M     ⟧c = η⟦A⟧t ∘ ⟦M⟧m
⟦ letEvt E In C  ⟧c = bindEvent Γ ⟦E⟧m ⟦C⟧c
⟦  select E1 ↦ C1 | E2 ↦ C3 | both ↦ C3  ⟧c =
    bindEvent Γ ⟪⟦E1⟧m, ⟦E2⟧m⟫ (handle ⟦C1⟧c ⟦C2⟧c ⟦C2⟧c)
```

Computations $\Gamma \vdash C \div A$ are interpreted as *Kleisli arrows* $[\![\Gamma]\!]_{\mathrm{x}} \mathrel{-\!\!\triangleright} \diamond[\![A]\!]_{\mathrm{j}}$, as we alluded to in Section 4.1.2. Pure computations use η to wrap the inner term into a $\diamond$. The `letSta` case is similar to the semantics of stable binding in terms. The denotation of events and selection both use a helper function `bindEvent`: it takes an event and an event handler and interprets them as a Kleisli arrow from the context to the return type of the body.

```
bindEvent : ∀ Γ {A B}
        -> (⟦Γ⟧x ⇸ ◇A) -> (⟦Γ ˢ⟧x ⊗ A ⇸ ◇B) -> (⟦Γ⟧x ⇸ ◇B)

bindEvent Γ {A}{B} E C = μB ∘ ◇(C ∘ ε⟦Γ⟧x × idA) ∘ st□⟦Γ⟧x,A ∘ ⟨⟦ˢ⟧□Γ, E⟩
```

The function first extends the stabilised context with the denotation of the event, then applies $\square$-tensorial strength to move the $\diamond$ from the event type to the whole context. Then, under the $\diamond$, we apply extraction to $\square[\![\Gamma^{\,s}]\!]_{\mathrm{x}}$ and interpret `C` in this context. Finally, we join the two diamonds with μ.

$$h = \square[\![\Gamma^{\,s}]\!]_{\mathrm{x}} \otimes \mathsf{A} \xrightarrow{\varepsilon_{[\![\Gamma]\!]_{\mathrm{x}}} \times \mathrm{id}_{\mathsf{A}}} [\![\Gamma^{\,s}]\!]_{\mathrm{x}} \otimes \mathsf{A} \xrightarrow{[\![\mathsf{C}]\!]_{\mathrm{c}}} \diamond\mathsf{B}$$

$$[\![\Gamma]\!]_{\mathrm{x}} \xrightarrow{\langle [\![^{s}]\!]\square_{\Gamma}, [\![\mathsf{E}]\!]_{\mathrm{m}}\rangle} \square[\![\Gamma^{\,s}]\!]_{\mathrm{x}} \otimes \diamond\mathsf{A} \xrightarrow{\mathrm{st}^{\square}_{[\![\Gamma]\!]_{\mathrm{x}},\mathsf{A}}} \diamond(\square[\![\Gamma^{\,s}]\!]_{\mathrm{x}} \otimes \mathsf{A}) \xrightarrow{\diamond h} \diamond\diamond\mathsf{B} \xrightarrow{\mu_B} \diamond\mathsf{B}$$

The interpretation of selection binds the linear-time product morphism of the two events in a body which selects the correct continuation. As linear-time products are sums of the three ordering possibilities, `handle` can case-split on the sum and return the suitable computation.

Note that `bindEvent` bears some resemblance to the monadic bind operator `>>=` – indeed, this is how we interpreted events originally. The semantics were revised in an attempt to provide a fully compositional interpretation of the language to highlight the categorical properties of **Reactive** required for a sound semantics. We conjecture the two approaches to be equivalent, though more work is needed to prove this formally.

5.1.3 Substitutions

In order to prove the soundness of our language, we need to establish that our substitution operators preserve the meaning of the terms. Expressed formally, soundness of substitution states that the denotation of a term M with an explicit substitution σ applied (which can encompass any context transformation we require) is the same as the denotation of σ followed by the denotation of M:

$$[\![[\sigma]M]\!]_{\mathrm{m}} = [\![M]\!]_{\mathrm{m}} \circ [\![\sigma]\!]_{\mathrm{s}}$$

As explicit substitutions are context transformations, their denotation should be a morphism between the interpretation of the contexts. This is a standard approach in categorical semantics

(Pitts, 2016), however, we are not aware of any previous attempts to integrate it with McBride's traversal framework, which we found to be a really useful syntactic formulation of term substitution. Therefore, we developed a categorical semantic analogue to syntactic kits in which every kit operation and substitution combinator is associated with a simple soundness lemma, and the soundness of the generic term travesal operation is established using these "proof combinators". Then, as with syntactic kits, the soundness of substitution, weakening, exchange, etc. are instances of the traversal soundness proof, specialised to an appropriate explicit substitution. As an example, below is the categorical proof that substitution for the top variable of the context is sound; again, the implementation details are left to the Appendix B.

```
⟦sub-top_s⟧ : ∀{Γ A} -> (M : Γ ⊢ A) -> ⟦sub-top_s M⟧_s = ⟨id_Γ, ⟦M⟧_m⟩
⟦sub-top_s⟧ M rewrite ⟦id_s⟧_s = refl

subst-sound : ∀{Γ A B} (M : Γ ⊢ A) (N : Γ, A ⊢ B)
           -> ⟦[M /] N⟧_m = ⟦N⟧_m ∘ ⟨id_Γ, ⟦M⟧_m⟩
subst-sound M N rewrite sym (⟦sub-top_s⟧ M) =
    substitute-sound (sub-top_s M) N
```

We first prove that the denotation of explicit substitution for the top of the context is equal to the product morphism $\langle \mathrm{id}_\Gamma, [\![M]\!]_m \rangle$ (using the proof that the interpretation of the identity substitution is the identity morphism), then use this to invoke the generic soundness lemma `substitute-sound` on top substitutions. We have similar lemmas for the two other kinds of substitution we defined. These are essential in proving the soundness of our categorical semantics, as demonstrated in the next section.

5.2 Soundness of term equality

We now have everything to prove that our categorical semantics is sound with respect to the term equality judgements introduced in Section 3.2. We can state this formally as:

Theorem (Soundness). *Let $\Gamma \vdash M, N : A$ be two terms of our language.*
If $\Gamma \vdash M \equiv N : A$, then $[\![M]\!]_m$, $[\![N]\!]_m$ are equal morphisms in the hom-set **Reactive**$([\![\Gamma]\!]_x, [\![A]\!]_j)$.

A similar soundness theorem can be stated for computations.

Proof. The proof is, for the most part, quite simple:

- The equivalence relation rules for term equality translate into the reflexive, symmetric and transitive properties of morphism equality.
- β-equality for pairs translates into definitionally equal morphisms; for reductions involving bound variables, we prove equality of morphisms using the soundness of substitution lemmas from the previous section.
- Soundness for η-expansion either holds definitionally (for units and stable types), by BCCC laws (pairs and sums), or monad laws (events).
- Soundness for congruences holds by recursively establishing the soundness of the assumption equality.

Below we show some cases of the proof.

```
sound : ∀{Γ A} {M N : Γ ⊢ A} -> Γ ⊢ M ≡ N :: A -> ⟦M⟧m = ⟦N⟧m
sound (Eq.refl M) = refl
sound (Eq.sym p) = sym (sound p)
sound (Eq.trans p q) = trans (sound p) (sound q)

sound (β-lam N M) rewrite subst-sound M N = refl
sound (β-fst M N) = refl
sound (β-inl M N _) rewrite subst-sound M N = refl
sound (β-sta N M) rewrite subst-sound M N = refl

sound (η-lam M) rewrite ⟦w⟧ M = refl
sound (η-pair M) = ⊗-η-exp ⟦M⟧m   -- A CCC lemma
sound (η-sta M) = refl

sound (cong-fst p) rewrite sound p = refl
sound (cong-app p q) rewrite sound p | sound q = refl
sound (cong-letSta p N) rewrite sound p = refl
sound (cong-event p) rewrite sound' p = refl

sound' : ∀{Γ A} {C D : Γ ⊨ A} -> Γ ⊨ C ≡ D :: A -> ⟦C⟧c = ⟦D⟧c
sound' (Eq'.trans p q) = trans (sound' p) (sound' q)
sound' (β-evt' C D) rewrite subst''-sound D C = refl
sound' (η-sta' M) = refl
sound' (cong-pure' p) rewrite sound p = refl
```

Thus, we conclude that our categorical semantics is sound with respect to term equality. □

CHAPTER 6

Conclusions

In this dissertation we described a small language for reactive programming, together with a sound denotational semantics in the category of reactive types. The syntax and semantics integrate previous approaches to strongly typed, efficient FRP, notably Jeffrey's and Jeltsch's linear temporal type systems and Krishnaswami's delay operator. The syntax – based on Pfenning and Davies' judgemental modal logic – was formalised in Agda, together with a syntactic framework for explicit substitutions and a $\beta\eta$-equational theory. We defined a concrete categorical model of linear temporal logic which would give a basis for an implementation that avoids resource waste due to polling. The syntax was given a categorical semantics in this model, with soundness proofs for explicit substitutions and term equality.

6.1 Related work

Most of the relevant research and historical developments were cited throughout the dissertation, notably in Chapter 2 and Section 4.1. Here we mention interesting lines of work that outline possible extensions to our system.

- Sculthorpe and Nilsson (2009) formalise the signal functions of Yampa (Hudak, Courtney, et al., 2002) in a dependently typed setting, extending it to address its safety and performance limitations.
- Jeffrey (2013) shows how his Agda implementation of reactive types can be compiled to JavaScript to enable dependently typed development of reactive web applications.
- Jeltsch (2014b) extends the notion of temporal categories to recursive abstract process categories that can handle FRP processes (corresponding to the $\mathcal{U}$ modality) with recursion.
- Paykin, Krishnaswami, and Zdancewic (2016) establish a connection between event-driven programming and temporal logic with linear extensions, and propose using callbacks to model logical negation. That way, the classical duality $\Diamond A = \neg\Box\neg A$ expresses the type $\Box(A \to \bot) \to \bot$ which bears close resemblance to the type of continuation-passing style functions.

6.2 Future work

In this project, we hope to set the foundation for a future line of FRP research – as such, there are some specific developments and improvements that we would like to add to the system.

- The semantics was made more abstract relatively late into the project, so some of the metatheory – notably the soundness proof of computation substitution – needs to be changed accordingly. While we believe that the two semantics are equivalent, this has not been formally proved yet.
- We need to revisit the syntax and semantics of `select`: as of now, it only has a single reduction rule instead of the expected three. We suspect that this will require adding a new explicit delay term to the syntax, but due to the nuances of computation substitution our initial attempt at this was unsuccessful.
- Krishnaswami (2013) showed how Nakano-style guarded recursion can encode various temporal types in a system similar to ours. We intend to add a similar temporal recursive type to our syntax, but use Event as the guard instead of $\bullet$.
- The next big step is proving that our definition of $\Diamond$ is implementable via continuation-passing style as $\Diamond A \approx \neg\Box\neg A \approx \Box(A \to \texttt{IO ()}) \to \texttt{IO ()}$: this is likely to require an operational semantics which can be developed from our equational theory.
- There are many other aspects that we could extend the system with: processes and the $\mathcal{U}$ modality, signal functions, linear types, etc.
- Ultimately, we hope to implement our language as a framework for efficient, statically correct functional reactive programming.

Bibliography

Abadi, Martin, Luca Cardelli, Pierre Louis Curien, and Jean Jacques Lévy (1991).
Explicit substitutions. In: *Journal of Functional Programming* 1.4, pp. 375–416.

Aczel, Peter, Jiří Adámek, Stefan Milius, and Jiří Velebil (2003).
Infinite trees and completely iterative theories: a coalgebraic view. In: *Theoretical Computer Science* 300.1-3, pp. 1–45.

Alechina, Natasha, Michael Mendler, Valeria de Paiva, and Eike Ritter (2001).
Categorical and Kripke Semantics for Constructive S4 Modal Logic. In: *Proceedings of the 15th International Workshop on Computer Science Logic*. Springer-Verlag, pp. 292–307.

Altenkirch, Thorsten, Martin Hofmann, and Thomas Streicher (1995).
Categorical reconstruction of a reduction free normalization proof. In: *Proceedings of the International Conference on Category Theory and Computer Science*, pp. 182–199.

Altenkirch, Thorsten and Bernhard Reus (1999).
Monadic Presentations of Lambda Terms Using Generalized Inductive Types. In: *Proceedings of the 13th International Workshop and 8th Annual Conference of the EACSL on Computer Science Logic*. Springer-Verlag, pp. 453–468.

Awodey, Steve (2010).
Category theory. Oxford University Press.

Bellegarde, Françoise and James Hook (1994).
Substitution: A formal methods case study using monads and transformations. In: *Science of Computer Programming* 23.2-3, pp. 287–311.

Bellin, Gianluigi, Valeria de Paiva, and Eike Ritter (2001).
Extended Curry-Howard correspondence for a basic constructive modal logic. In: *In Proceedings of Methods for Modalities*.

Benton, Nick, Gavin M. Bierman, and Valeria de Paiva (1998).
Computational types from a logical perspective. In: *Journal of Functional Programming* 8.2, pp. 177–193.

Bierman, Gavin M. and Valeria de Paiva (2000).
On an Intuitionistic Modal Logic. In: *Studia Logica* 65.3, pp. 383–416.

Bird, Richard S. and Ross Paterson (1999).
De Bruijn notation as a nested datatype. In: *Journal of Functional Programming* 9.1, pp. 77–91.

Cave, Andrew, Francisco Ferreira, Prakash Panangaden, and Brigitte Pientka (2014).
Fair reactive programming. In: *ACM SIGPLAN Notices*. Vol. 49. 1, pp. 361–372.

Chapman, James Maitland (2009).
Type checking and normalisation. PhD thesis. University of Nottingham.

Clarke, Edmund M. and E. Allen Emerson (1981).
Design and synthesis of synchronization skeletons using branching time temporal logic. In: *Proceedings of the Workshop on Logic of Programs*, pp. 52–71.

Crole, Roy L. (1993).
Categories for types. Cambridge University Press.

Curry, Haskell B. (1952).
The elimination theorem when modality is present. In: *Journal of Symbolic Logic* 17.4, pp. 249–265.

Czaplicki, Evan and Stephen Chong (2013).
Asynchronous Functional Reactive Programming for GUIs. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 411–422.

de Bruijn, Nicolaas Govert (1972).
Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In: *Indagationes Mathematicae (Proceedings)*. Vol. 75. 5, pp. 381–392.

de Paiva, Valeria and Harley Eades III (2017).
Constructive Temporal Logic, Categorically. In: *Journal of Logic and its Applications* 4.4. Special Issue Dedicated to the Memory of Grigori Mints.

de Paiva, Valeria and Eike Ritter (2016).
Fibrational Modal Type Theory. In: *Electronic Notes in Theoretical Computer Science* 323, pp. 143–161.

Dybjer, Peter (1994).
Inductive families. In: *Formal Aspects of Computing* 6.4, pp. 440–465.

Elliott, Conal (2009a).
Denotational design with type class morphisms (extended version). Tech. rep. 2009-01. LambdaPix.

Elliott, Conal (2009b).
Push-pull functional reactive programming. In: *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pp. 25–36.

Elliott, Conal (2015).
The essence and origins of FRP. `http://conal.net/talks/essence-and-origins-of-frp-bayhac-2015.pdf`.

Elliott, Conal and Paul Hudak (1997).
Functional reactive animation. In: *ACM SIGPLAN Notices*. Vol. 32. 8, pp. 263–273.

Fairtlough, Matt and Michael Mendler (1994).
An intuitionistic modal logic with applications to the formal verification of hardware. In: *Proceedings of the International Workshop on Computer Science Logic*, pp. 354–368.

Goguen, Healfdene and James McKinna (1997).
Candidates for substitution. In: *Laboratory for Foundations of Computer Science – Report Series*.

Hardy, Bradley (2017).
Better Equational Reasoning for Agda. Undergraduate dissertation. University of Cambridge.

Harper, Robert (2011).
The Holy Trinity. `https://existentialtype.wordpress.com/2011/03/27/the-holy-trinity/`. Post in the *Existential Type* blog.

Howard, William A. (1980).
The formulae-as-types notion of construction. In: *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism* 44, pp. 479–490.

Hudak, Paul, Antony Courtney, Henrik Nilsson, and John Peterson (2002).
Arrows, robots, and functional reactive programming. In: *International School on Advanced Functional Programming*, pp. 159–187.

Hudak, Paul, John Hughes, Simon Peyton Jones, and Philip Wadler (2007).
A history of Haskell: being lazy with class. In: *Proceedings of the 3rd ACM SIGPLAN Conference on History of Programming Languages*, pp. 12–1.

Hughes, John (2000).
Generalising monads to arrows. In: *Science of Computer Programming* 37.1-3, pp. 67–111.

Jeffrey, Alan (2012).
LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In: *Proceedings of the 6th Workshop on Programming Languages meets Program Verification*, pp. 49–60.

Jeffrey, Alan (2013).
Dependently Typed Web Client Applications. In: *Proceedings of the 15th International Symposium on Practical Aspects of Declarative Languages*, pp. 228–243.

Jeffrey, Alan (2014).
Functional reactive types. In: *Proceedings of the Joint Meeting of the 23rd Conference on Computer Science Logic and the 29th Symposium on Logic in Computer Science*, p. 54.

Jeltsch, Wolfgang (2011).
Strongly typed and efficient functional reactive programming. PhD thesis. Brandenburg University of Technology Cottbus-Senftenberg.

Jeltsch, Wolfgang (2012).
Towards a Common Categorical Semantics for Linear-Time Temporal Logic and Functional Reactive Programming. In: *Electronic Notes in Theoretical Computer Science* 286, pp. 229–242.

Jeltsch, Wolfgang (2013).
Temporal logic with "Until", functional reactive programming with processes, and concrete process categories. In: *Proceedings of the 7th Workshop on Programming Languages meets Program Verification*, pp. 69–78.

Jeltsch, Wolfgang (2014a).
An abstract categorical semantics for functional reactive programming with processes. In: *Proceedings of the 8th Workshop on Programming Languages meets Program Verification*, pp. 47–58.

Jeltsch, Wolfgang (2014b).
Categorical Semantics for Functional Reactive Programming with Temporal Recursion and Corecursion. In: *arXiv.org*, pp. 127–142. arXiv: `1406.2062v1`.

Keller, Chantal (2008).
The category of simply typed λ-terms in Agda. Available at `https://www.lri.fr/~keller/Documents-recherche/Stage08/Parallel-substitution/report.pdf`.

Kobayashi, Satoshi (1997).
Monad as modality. In: *Theoretical Computer Science* 175.1, pp. 29–74.

Kock, Anders (1972).
Strong functors and monoidal monads. In: *Archiv der Mathematik* 23.1, pp. 113–120.

Krishnaswami, Neelakantan R. (2013).
Higher-order Functional Reactive Programming Without Spacetime Leaks. In: *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 221–232.

Krishnaswami, Neelakantan R. and Nick Benton (2011a).
A semantic model for graphical user interfaces. In: *ACM SIGPLAN Notices*. Vol. 46. 9, pp. 45–57.

Krishnaswami, Neelakantan R. and Nick Benton (2011b).
Ultrametric semantics of reactive programs. In: *Proceedings of the 26th Annual IEEE Symposium on Logic in Computer Science*, pp. 257–266.

Krishnaswami, Neelakantan R., Nick Benton, and Jan Hoffmann (2012).
Higher-order functional reactive programming in bounded space. In: *ACM SIGPLAN Notices*. Vol. 47. 1, pp. 45–58.

Lambek, Joachim (1980).
From lambda-calculus to cartesian closed categories. In: *To H.B.Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pp. 375–402.

Lawvere, F. William (1964).
An elementary theory of the category of sets. In: *Proceedings of the National Academy of Sciences* 52.6, pp. 1506–1511.

Mac Lane, Saunders (1978).
Categories for the Working Mathematician. Springer New York.

Martin-Löf, Per (1998).
An intuitionistic theory of types. In: *Twenty-five years of constructive type theory* 36, pp. 127–172.

McBride, Conor (2004).
Epigram: Practical programming with dependent types. In: *Proceedings of the International School on Advanced Functional Programming*, pp. 130–170.
McBride, Conor (2005).
Type-preserving renaming and substitution. Available at `http://strictlypositive.org/ren-sub.pdf`.
Moggi, Eugenio (1991).
Notions of computation and monads. In: *Information and Computation* 93.1, pp. 55–92.
Nakano, Hiroshi (2000).
A modality for recursion. In: *Proceedings of the 15th Symposium on Logic in Computer Science*, pp. 255–266.
Nilsson, Henrik, Antony Courtney, and John Peterson (2002).
Functional reactive programming, continued. In: *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pp. 51–64.
Norell, Ulf (2008).
Dependently typed programming in Agda. In: *Proceedings of the International School on Advanced Functional Programming*, pp. 230–266.
Paterson, Ross (2001).
A New Notation for Arrows. In: *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 229–240.
Paykin, Jennifer, Neelakantan R. Krishnaswami, and Steve Zdancewic (2016).
The Essence of Event-Driven Programming. Available at `https://www.cl.cam.ac.uk/~nk480/essence-of-events.pdf`. Draft.
Pfenning, Frank and Rowan Davies (2001).
A judgmental reconstruction of modal logic. In: *Mathematical Structures in Computer Science* 11.4.
Pierce, Benjamin C (2005).
Advanced topics in types and programming languages. MIT Press.
Pitts, Andrew (2016).
Brief Notes on the Category Theoretic Semantics of Simply Typed Lambda Calculus. `https://www.cl.cam.ac.uk/teaching/1617/L108/catl-notes.pdf`.
Pnueli, Amir (1977).
The temporal logic of programs. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46–57.
Prawitz, Dag (1965).
Natural deduction: a proof-theoretical study. PhD thesis. Almqvist & Wiksell.
Quick, Donya and Paul Hudak (2013).
Grammar-based automated music composition in Haskell. In: *Proceedings of the 1st ACM SIGPLAN workshop on Functional Art, Music, Modeling & Design*, pp. 59–70.
Sculthorpe, Neil and Henrik Nilsson (2009).
Safe Functional Reactive Programming Through Dependent Types. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*. ACM, pp. 23–34.
Wadler, Philip (1995).
Monads for functional programming. In: *Proceedings of the International School on Advanced Functional Programming*, pp. 24–52.
Wan, Zhanyong, Walid Taha, and Paul Hudak (2002).
Event-driven FRP. In: *Proceedings of the International Symposium on Practical Aspects of Declarative Languages*, pp. 155–172.

APPENDIX A

Category theory

This section presents a brief overview of the core concepts of category theory. It is intended as a place to establish the terminology and notation used in the dissertation, not as an exhaustive account of the topic – for the details, there are many great textbooks to refer to, such as the classic foundational text of Mac Lane (1978) or the more contemporary introduction by Awodey (2010).

Bicartesian closed categories

In this section, we define the basic notions of categories, products, coproducts, exponentials and bicartesian closed categories.

Definition (Category). A *category* $\mathbb{C}$ consists of:

- a collection of objects $\mathrm{Ob}(\mathbb{C})$;
- for each pair of objects $A, B \in \mathrm{Ob}(\mathbb{C})$, a collection of *morphisms* $\mathbb{C}(A, B)$ called the *hom-set* of A and B;
- for every object A, a (unique) *identity morphism* $\mathrm{id}_A \colon A \to A$;
- for every pair of morphisms $f \colon A \to B$ and $g \colon B \to C$, a *composition* $g \circ f \colon A \to C$ obeying the following laws:
  - the identity morphism is a right and left identity for composition: $f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f$;
  - composition is associative: for all other $h \colon C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f = h \circ g \circ f$.

One of the core ideas of category theory is *universal constructions*: defining concepts by giving the properties it must obey, alongside with a way to select the "best" construction from all the ones which obey these properties. Examples of universal constructions are limits and colimits, and below we define some special cases.

Definition (Terminal object). An object $\top$ in a category $\mathbb{C}$ is a *terminal object* if for all $A \in \mathbb{C}$ there exists a unique morphism $!_A \colon A \to \top$.

Definition (Product object). For two objects $A, B \in \mathbb{C}$, the *product* $A \times B$ (if it exists) is an object with two *projections* $\pi_1 \colon A \times B \to A$ and $\pi_2 \colon A \times B \to B$ such that for any other candidate product

P with morphisms $p_1 \colon P \longrightarrow A$ and $p_2 \colon P \longrightarrow B$ there exists a unique *mediator* $m \colon P \longrightarrow A \times B$ such that $p_1 = \pi_1 \circ m$ and $p_2 = \pi_2 \circ m$.

The usual way to summarise these definitions graphically is via a *commutative diagram*:

$$\begin{tikzcd}
 & P \arrow[d, dashed, "m"] \arrow[ddl, bend right, "p_1"'] \arrow[ddr, bend left, "p_2"] & \\
 & A \times B \arrow[dl, "\pi_1"'] \arrow[dr, "\pi_2"] & \\
A & & B
\end{tikzcd}$$

Products also have associated morphism operations:

Definition. Given two morphisms $f \colon P \longrightarrow A$ and $g \colon P \longrightarrow B$ with the same source, we can form the *product morphism* $\langle f, g \rangle \colon P \longrightarrow A \times B$, defined as the unique mediator between the candidate product P and $A \times B$.
For morphisms $f \colon P \longrightarrow A$ and $g \colon Q \longrightarrow B$, their *parallel product* $f \times g \colon P \times Q \longrightarrow A \times B$ is defined as $f \times g = \langle f \circ \pi_1, g \circ \pi_2 \rangle$.

Next, we define the *duals* of terminal and product objects, i.e. terminal objects and products in the category $\mathbb{C}^{\mathrm{op}}$ with all arrows reversed.

Definition (Initial object). An object $\bot$ in a category $\mathbb{C}$ is an *initial object* if for all $A \in \mathbb{C}$ there exists a unique morphism $¡_A \colon \bot \longrightarrow A$.

Definition (Sum object). For two objects $A, B \in \mathbb{C}$, the *sum* (or coproduct) $A + B$ (if it exists) is an object with two *injections* $\iota_1 \colon A \longrightarrow A + B$ and $\iota_2 \colon B \longrightarrow A + B$ such that for any other candidate sum S with morphisms $i_1 \colon A \longrightarrow S$ and $i_2 \colon B \longrightarrow S$ there exists a unique *mediator* $m \colon A + B \longrightarrow S$ such that $s_1 = m \circ \iota_1$ and $s_2 = m \circ \iota_2$.

$$\begin{tikzcd}
A \arrow[dr, "\iota_1"] \arrow[ddr, bend right, "i_1"'] & & B \arrow[dl, "\iota_2"'] \arrow[ddl, bend left, "i_2"] \\
 & A + B \arrow[d, dashed, "m"] & \\
 & S &
\end{tikzcd}$$

Definition. Given two morphisms $f \colon A \longrightarrow S$ and $g \colon B \longrightarrow S$ with the same target, we can form the *sum morphism* $[f, g] : A + B \longrightarrow P$, defined as the unique mediator between $A \times B$ and the candidate sum P.
For morphisms $f \colon P \longrightarrow A$ and $g \colon Q \longrightarrow B$, their *parallel sum* $f + g \colon P + Q \longrightarrow A + B$ is defined as $f + g = [\iota_1 \circ f, \iota_2 \circ g]$.

As expected, products and sums generalise the set-theoretic Cartesian product and disjoint union operations, which are the products and coproducts in the category **Set** of sets. We can also introduce a third kind of object which internalises morphisms – this is analogous to how the functions between two sets (i.e. the hom-sets in **Set**) are themselves sets.

Definition (Exponential object). For two objects A, B in a category $\mathbb{C}$ with all binary products, their *exponential* $A \Rightarrow B$ or B^A (if it exists) is an object with a morphism $\mathrm{ev}\colon (A \Rightarrow B) \times A \to B$ such that for any other candidate exponential E with morphism $e\colon E \times A \to B$ there exists a unique mediator $m\colon E \to (A \Rightarrow B)$ such that the following diagram commutes:

$$\begin{tikzcd}
E \arrow[d, dashed, "m"] & E \times A \arrow[d, dashed, "m \times \mathrm{id}_A"'] \arrow[dr, "e"] & \\
A \Rightarrow B & (A \Rightarrow B) \times A \arrow[r, "\mathrm{ev}"'] & B
\end{tikzcd}$$

Definition. Given a morphism $f : E \times A \to B$, its *currying* or *transpose* $\Lambda f : E \to (A \Rightarrow B)$ is defined as the unique mediator between the candidate exponential E and $A \Rightarrow B$.

We are now ready to give the definition of an important class of categories which can be used to model various computational calculi, including our language.

Definition. A *Cartesian closed category* (CCC) is a category $\mathbb{C}$ which is:

- *Cartesian*, i.e. has all finite products (or equivalently, has a terminal object and a product $A \times B$ for every pair of objects $A, B \in \mathbb{C}$);
- *closed*, i.e. has an exponential $A \Rightarrow B$ for every pair of objects $A, B \in \mathbb{C}$.

A *bicartesian closed category* (BCCC) is a cartesian closed category with all finite sums.

Functors and natural transformations

Informally, functors are maps between categories, and natural transformations are maps between functors.

Definition (Functors). A *functor* F between two categories $\mathbb{C}$ and $\mathbb{D}$ consists of:

- an object map $F_\mathrm{o}\colon \mathrm{Ob}(\mathbb{C}) \to \mathrm{Ob}(\mathbb{D})$;
- for every object $A, B \in \mathbb{C}$, a morphism map $F_\mathrm{m}\colon \mathbb{C}(A, B) \to \mathbb{D}(F_\mathrm{o}(A), F_\mathrm{o}(B))$ which obeys the following properties:
  - it preserves identities: for all $A \in \mathbb{C}$, $F_\mathrm{m}(\mathrm{id}^{\mathbb{C}}_A) = \mathrm{id}^{\mathbb{D}}_{F_\mathrm{o}(A)}$;
  - it preserves composition: for all objects $A, B, C \in \mathbb{C}$ and morphisms $f\colon A \to B$, $g\colon B \to C$, $F_\mathrm{m}(g \circ^{\mathbb{C}} f) = F_\mathrm{o}(g) \circ^{\mathbb{D}} F_\mathrm{o}(f)$.

Usually both F_o and F_m are denoted by F: as they act on different entities (objects FA or morphisms Ff), there is rarely any ambiguity. As functors are a kind of mapping, we might wonder if they are the morphisms in any category – sure enough, they are morphisms in the category of categories, **Cat**, whose objects are (small) categories and morphisms are functors. While we will not need this category explicitly, we will use the characteristic identity functors $\mathrm{Id}_{\mathbb{C}}$ and functor composition $GF\colon \mathbb{C} \to \mathbb{E}$ for $F\colon \mathbb{C} \to \mathbb{D}$ and $G\colon \mathbb{D} \to \mathbb{E}$.

Functors can be used to compare or transform categories, but we are often interested in comparing ways to transform categories. This is where natural transformations come into play.

Definition (Natural transformation). A *natural transformation* $\phi\colon F \Longrightarrow G$ between two functors $F, G\colon \mathbb{C} \to \mathbb{D}$ is a collection of morphisms in $\mathbb{D}$ indexed by objects in $\mathbb{C}$. A *component* of ϕ at $A \in \mathbb{C}$, denoted ϕ_A, is a morphism from FA to GA, i.e. an element of $\mathbb{D}(FA, GA)$. The natural transformation must obey the *naturality condition* for every morphism $f\colon A \to B$:

$$Gf \circ \phi_A = \phi_B \circ Ff$$

This can also be expressed as a *naturality square.* Natural transformations then map objects in $\mathbb{C}$ to morphisms in $\mathbb{D}$, and morphisms in $\mathbb{C}$ to naturality squares in $\mathbb{D}$.

$$\begin{tikzcd}
A \arrow[d, "\forall f"'] & FA \arrow[r, "\phi_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"] \\
B & FB \arrow[r, "\phi_B"'] & GB
\end{tikzcd}$$

As before, it can be shown that functors between two categories $\mathbb{C}$ and $\mathbb{D}$ themselves form a category called a *functor category* $\mathbb{D}^{\mathbb{C}}$: its objects are functors $F, G\colon \mathbb{C} \to \mathbb{D}$ and morphisms are natural transformations $F \Longrightarrow G$. We can define the identity natural transformation id_F and (vertical) composition of natural transformations $\psi \circ \phi$ componentwise, and prove their coherence properties.

The last three concepts relate functors and natural transformations, and they will be very important for our discussion of modal logic in category theory.

Adjunctions, monads and comonads

An adjunction is a very general connection between two functors, expressing a weak form of an inverse relationship. Adjoint functors appear very frequently throughout mathematics, which makes them one of the most powerful tools in a mathematician's arsenal. They are also closely related to monads and comonads which have been embraced by computer science to formalise the notion of a computation and contexts.

Definition (Adjunction). Let $F\colon \mathbb{D} \to \mathbb{C}$ and $G\colon \mathbb{C} \to \mathbb{D}$ be two functors between categories $\mathbb{C}$ and $\mathbb{D}$. The functor F is *left adjoint* to G (equivalently, G is *right-adjoint* to F), denoted as $F \dashv G$, if:

- there exists a natural transformation $\eta\colon \mathrm{Id}_{\mathbb{D}} \Longrightarrow GF$ called the *unit*;
- there exists a natural transformation $\varepsilon\colon FG \Longrightarrow \mathrm{Id}_{\mathbb{C}}$ called the *counit*;
- the unit and counit satisfy the *triangle identities*:

$$\varepsilon F \circ F\eta = \mathrm{id}_F \qquad G\varepsilon \circ \eta G = \mathrm{id}_G$$

$$\begin{tikzcd}
F \arrow[d, "F\eta"'] \arrow[dr, equal, "\mathrm{id}_F"] & \\
FGF \arrow[r, "\varepsilon F"'] & F
\end{tikzcd}
\qquad
\begin{tikzcd}
G \arrow[d, "\eta G"'] \arrow[dr, equal, "\mathrm{id}_G"] & \\
GFG \arrow[r, "G\varepsilon"'] & G
\end{tikzcd}$$

The above equalities and diagrams are given from the "point of view" of functor categories – they are actually shorthands for the component-wise equations (for $A \in \mathbb{C}$ and $B \in \mathbb{D}$)

$$\varepsilon_{FB} \circ F(\eta_B) = \mathrm{id}_{FB} \qquad\qquad G(\varepsilon_A) \circ \eta_{GA} = \mathrm{id}_{GA}$$

The first notation is preferred, as it lets us manipulate functors and natural transformations in an intuitive "algebraic" manner.

There are mahy examples for adjunctions, and their utility will become even more apparent when we relate them with monads and comonads.

Definition (Monad). Given a category $\mathbb{C}$, an endofunctor $T\colon \mathbb{C} \to \mathbb{C}$ is a *monad* if:

- there exists a natural transformation $\eta\colon \mathrm{Id}_{\mathbb{C}} \Longrightarrow T$ called *unit*;
- there exists a natural transformation $\mu\colon T^2 \Longrightarrow T$ called *multiplication*;
- unit and multiplication satisfy the following monad laws:
  - left and right unit: $\mu \circ T\eta = \mu \circ \eta T = \mathrm{id}_T$
  - associativity: $\mu \circ T\mu = \mu \circ \mu T$

$$\begin{tikzcd}
T \arrow[r, "T\eta"] \arrow[dr, equal, "\mathrm{id}_T"'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "\eta T"'] \arrow[dl, equal, "\mathrm{id}_T"] \\
& T &
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3 \arrow[r, "T\mu"] \arrow[d, "\mu T"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

Definition (Comonad). Given a category $\mathbb{C}$, an endofunctor $U\colon \mathbb{C} \to \mathbb{C}$ is a *comonad* if:

- there exists a natural transformation $\varepsilon\colon U \Longrightarrow \mathrm{Id}_{\mathbb{C}}$ called *counit* or *extraction*;
- there exists a natural transformation $\delta\colon U \Longrightarrow U^2$ called *comultiplication* or *duplication*;
- counit and comultiplication satisfy the following comonad laws:
  - left and right counit: $U\varepsilon \circ \delta = \varepsilon U \circ \delta = \mathrm{id}_U$
  - coassociativity: $U\delta \circ \delta = \delta U \circ \delta$

$$\begin{tikzcd}
& U \arrow[d, "\delta"] \arrow[dl, equal, "\mathrm{id}_U"'] \arrow[dr, equal, "\mathrm{id}_U"] & \\
U & U^2 \arrow[l, "U\varepsilon"] \arrow[r, "\varepsilon U"'] & U
\end{tikzcd}
\qquad
\begin{tikzcd}
U \arrow[r, "\delta"] \arrow[d, "\delta"'] & U^2 \arrow[d, "U\delta"] \\
U^2 \arrow[r, "\delta U"'] & U^3
\end{tikzcd}$$

As it turns out, every adjunction induces a monad-comonad pair (and every monad and comonad can be derived from an adjunction).

Theorem. *Let $F\colon \mathbb{D} \to \mathbb{C}$ and $G\colon \mathbb{C} \to \mathbb{D}$ be two adjoint functors $F \dashv G$, with unit η and counit ε. Then:*

- *the functor $T = GF\colon \mathbb{D} \to \mathbb{D}$ is a monad, with unit η and multiplication $G\varepsilon F$;*
- *the functor $U = FG\colon \mathbb{C} \to \mathbb{D}$ is a comonad, with counit ε and comultiplication $F\eta G$.*

Proof. The monad and comonad laws follow directly from the triangle identities of $F \vdash G$ and naturality of ε and η. □

APPENDIX B

Substitution

Term substitution (e.g. an operation of type $\Gamma \vdash M : A \to \Gamma, x : A \vdash N : B \to \Gamma \vdash [M/x]N : B$) is easy to describe informally ("substitute the term M for every occurrence of x in N"), but actually defining it is remarkably complicated. We need to check for variable equality, handle renaming, avoid variable capture, and do everything as efficiently as possible. The de Bruijn indexing solves the first two issues but requires manipulating the indices to avoid capture, which can get quite cumbersome. Moreover, it is awkward to define substitution in our Agda formalisation "from first principles": we either get hopelessly lost trying to derive the variable and stabilisation cases, or terms involving bound variables.

In this section we present two approaches to solving these issues, which combine to create a general, intuitive and efficient framework for managing substitution and other syntactic meta-operations. In fact, we show that any context lemma – such as weakening, exchange or substitution – can be expressed as a term traversal operation with an appropriate substitution. We then develop a semantic version of this framework, which allows us to prove the soundness of these operations in a similarly generic way.

Syntactic kits

One half of our syntactic framework was first described by McBride (2005) in an unpublished Functional Pearl. It starts with the observation that in the Altenkirch and Reus style of term encoding, the terms and variables are both indexed by the same entities: a context and a judgement. In addition, the operations of variable renaming and term substitution can both be formulated as term traversals, mapping the free variables of a term either to another variable, or a term.

$$\Gamma, z\colon A \vdash \lambda f.\, \lambda y.\, f\, z\, y\, z \xleftarrow{\text{rename } x \text{ to } z} \Gamma, x\colon A \vdash \lambda f.\, \lambda y.\, f\, x\, y\, x \xrightarrow{\text{substitute } M \text{ for } x} \Gamma \vdash \lambda f.\, \lambda y.\, f\, M\, y\, M$$

$$\begin{aligned} \text{rename} :&\quad (A \in \Gamma \to A \in \Delta) \to (\Gamma \vdash A \to \Delta \vdash A) \\ \text{substitute} :&\quad (A \in \Gamma \to \Delta \vdash A) \to (\Gamma \vdash A \to \Delta \vdash A) \end{aligned}$$

In fact, it is possible to define a generic term traversal operation which takes a map from variables to "stuff" (McBride's terminology) and applies this map to each free variable of the term. This "stuff" – indexed by a context and a judgement – can be instantiated with a variable or a term, and the resulting traversal will either act as a renaming or a substitution.

$$\text{traverse}: \quad (A \in \Gamma \to \mathcal{S}\ \Delta\ A) \to (\Gamma \vdash A \to \Delta \vdash A)$$

Here we will call this "stuff" $\mathcal{S}$ a *schema*, and instances of a particular schema (i.e. variables or terms) $\mathcal{S}$-instances. McBride identifies three conditions that $\mathcal{S}$-instances must satisfy in order to allow us to define this generic traversal function. We should be able to turn variables into the schema, and the schema into a term. In addition, the schema should have a *weakening* map, i.e. a way to extend the context by a new type. These conditions can be collected into a so-called *syntactic kit*, which is a fairly vague name for a fairly vague – but nevertheless very powerful – concept.

Though McBride's paper used Epigram (McBride, 2004), the idea can be readily translated into Agda, as shown by Keller (2008). The original formulation was tailored to the λ-calculus, but – somewhat surprisingly – there is only one operation that we need to add to the kit to make it work with our system: mapping a schema $\mathcal{S}\ \Gamma\ (A \text{ always})$ to $\mathcal{S}\ \Gamma^{s}\ (A \text{ always})$. This makes intuitive sense: if something has a static type, its context can be stabilised. We now give the full specification of syntactic kits as an Agda record:

```
Schema : Set₁
Schema = Context -> Judgement -> Set
Var : Schema
Var = flip _∈_
Term : Schema
Term = _⊢_

record Kit (𝒮 : Schema) : Set where
  field
    𝓋 : ∀{Γ A}   -> A ∈ Γ -> 𝒮 Γ A
    𝓉 : ∀{Γ A}   -> 𝒮 Γ A -> Γ ⊢ A
    𝓌 : ∀{Γ A B} -> 𝒮 Γ A -> 𝒮 (Γ , B) A
    𝒶 : ∀{Γ A}   -> 𝒮 Γ (A always) -> 𝒮 (Γˢ) (A always)
```

Explicit substitutions

The main idea behind *explicit substitutions* (Abadi et al., 1991) is treating substitutions not as a term operation, but as explicit transformations between contexts (which Goguen and McKinna (1997) call context morphisms): a substitution $\Delta \vdash \sigma \rhd \Gamma$ changes the context of a schema (term or variable) $T : \mathcal{S}\ \Gamma\ A$ to $[\sigma]T : \mathcal{S}\ \Delta\ A$. By making substitutions a first-class syntactic entity (like in the $\lambda\sigma$-calculi originated by Abadi et al.), we can define new substitutions via *combinators*, without having to deal with the intricacies of the de Bruijn indexing. This also improves the efficiency of implementations, as substitutions can be "batched" and applied at the same time, potentially avoiding a term size explosion. Explicit substitutions can be nicely expressed in Agda and integrate well with syntactic kits.

In essence, a substitution is just a list of (variable, term) pairs alongside some well-formedness rules, and applying a substitution amounts to replacing the free variables in the argument with the corresponding term. Formally, we inductively define $\Delta \vdash \sigma_{\mathcal{S}} \triangleright \Gamma$ to represent a valid substitution of $\mathcal{S}$-instances in context Δ for the free variables of an $\mathcal{S}$-instance in context Γ. In the base case, we can substitute anything into a term with no free variables. In the inductive case, given a substitution $\Delta \vdash \sigma_{\mathcal{S}} \triangleright \Gamma$ and a $\mathcal{S}$-instance $T : \mathcal{S}\ \Delta\ A$, we can extend Γ by a new free variable for which we substitute T.

$$\frac{}{\Delta \vdash \bullet \triangleright \cdot} \qquad \frac{\Delta \vdash \sigma_{\mathcal{S}} \triangleright \Gamma \qquad T : \mathcal{S}\ \Delta\ A}{\Delta \vdash \sigma_{\mathcal{S}} \blacktriangleright T \triangleright \Gamma, A}$$

In Agda, this can be defined as an inductive family `Subst`, which guarantees that all substitutions are well-formed[1]:

```
data Subst (𝓢 : Schema) : Context -> Context -> Set where
  •    : ∀{Δ}      -> Subst 𝓢 · Δ
  _▶_  : ∀{A Γ Δ} -> Subst 𝓢 Γ Δ -> 𝓢 Δ A -> Subst 𝓢 (Γ , A) Δ
```

We now extend syntactic kits by adding the fundamental substitution property (as we will see later, it cannot be added to the `Kit` record directly):

```
record SubstKit (𝓢 : Schema) : Set where
  field
    𝓀 : Kit 𝓢
    𝓈 : ∀{Γ Δ A} -> Subst 𝓢 Γ Δ -> 𝓢 Γ A -> 𝓢 Δ A
```

Next, we define the substitution combinators for weakening, lifting and stabilisation, which are parameterised by a kit and use its 𝓌, 𝓋 and 𝒶 operations respectively.

```
_⁺_  : ∀{𝓢 Γ Δ A} -> Subst 𝓢 Γ Δ -> Kit 𝓢 -> Subst 𝓢 Γ (Δ, A)
_↑_  : ∀{𝓢 Γ Δ A} -> Subst 𝓢 Γ Δ -> Kit 𝓢 -> Subst 𝓢 (Γ, A) (Δ, A)
_↓ˢ_ : ∀{𝓢 Γ Δ}   -> Subst 𝓢 Γ Δ -> Kit 𝓢 -> Subst 𝓢 (Γˢ) (Δˢ)
```

As it turns out, explicit substitutions form the morphisms in the category of contexts (Keller, 2008), so we also provide identity and composition combinators – note that composition requires the substitution property of `SubstKit`:

```
idₛ      : ∀{𝓢 Γ} -> Kit 𝓢 -> Subst 𝓢 Γ Γ
_∘[_]ₛ_ : ∀{𝓢 Γ Δ Ξ} -> Subst 𝓢 Δ Ξ -> SubstKit 𝓢
                      -> Subst 𝓢 Γ Δ -> Subst 𝓢 Γ Ξ
```

Structural properties

The real pay-off when using explicit substitutions is that they can express any context transformation property in a very high-level way. Examples are not only substitutions, but a more

[1]Note that the order of contexts in the Agda definition may seem a bit backwards: while the formal definition of substitutions is consistent with the judgemental model (with the judgement $\sigma_{\mathcal{S}} \triangleright \Gamma$ expressing that "$\sigma_{\mathcal{S}}$ has the environment Γ"), it seemed sensible to notate a transformation from `𝓢 Γ A` to `𝓢 Δ A` as `Subst 𝓢 Γ Δ`.

general class of properties called *structural rules* (Pierce, 2005, Section 1.1) such as *weakening*, *exchange* and *contraction*:

$$\frac{\Gamma, \Delta \vdash A}{\Gamma, B, \Delta \vdash A}\ \text{(weaken)} \qquad \frac{\Gamma, A, \Delta, B, \Xi \vdash C}{\Gamma, B, \Delta, A, \Xi \vdash C}\ \text{(exchange)} \qquad \frac{\Gamma, A, A, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}\ \text{(contract)}$$

Our framework lets us express these lemmas as explicit substitutions, without ever referring to the actual term syntax of the language. Instead of having to define every lemma case-by-case (which gets even more problematic if we add a new term to the syntax), we declare them as context transformations which can then be applied in a term traversal. Below are the simplest versions of the lemmas, specialised to the top of the context; note how we are only using our syntactic kit functions and substitution combinators.

```
weak-topₛ : ∀{𝒮 Γ A} -> SubstKit 𝒮 -> Subst 𝒮 Γ (Γ, A)
weak-topₛ k = idₛ k ⁺ k

ex-topₛ : ∀{𝒮 Γ A B} -> SubstKit 𝒮 -> Subst 𝒮 (Γ, A, B) (Γ, B, A)
ex-topₛ k = (idₛ k ⁺ k ↑ k) ► (𝓋 k (pop top))

contr-topₛ : ∀{𝒮 Γ A} -> SubstKit 𝒮 -> Subst 𝒮 (Γ, A, A) (Γ, A)
contr-topₛ k = (idₛ k) ► (𝓋 k top)

sub-topₛ : ∀{𝒮 Γ A} -> SubstKit 𝒮 -> 𝒮 Γ A -> Subst 𝒮 (Γ, A) Γ
sub-topₛ k T = idₛ k ► T
```

We are now ready to define generic term traversals, provide `SubstKit` instances for variables and terms, and derive the structural and substitution lemmas we need.

Generic term traversal

Let us restate our main goal: defining a function that can apply an explicit substitution to a term.

```
substitute : ∀{Γ Δ A} -> Subst Term Γ Δ -> Γ ⊢ A -> Δ ⊢ A
```

This is actually the 𝓈 function of the `SubstKit` instance for `Term`. Arriving to this instance requires several steps, of which we highlight the two main ones.

Generic variable substitution map Recall from that McBride used a variable map of type $A \in \Gamma \rightarrow \mathcal{S}\ \Delta\ A$ as an argument to the traverse function: during a traversal, this map gets applied to the free variables of the term. We can define a function `subst-var` to turn any substitution into a map of this type. This is precisely the bridge between explicit substitutions and McBride's syntactic kit traversals.

```
subst-var : ∀{𝒮 Γ Δ A} -> Subst 𝒮 Γ Δ -> (Var Γ A -> 𝒮 Δ A)
subst-var • ()  -- Impossible pattern
```

```
subst-var (σ ► T) top     = T
subst-var (σ ► T) (pop v) = subst-var σ v
```

When specialised to variables, this function lets us provide a `SubstKit` instance $\mathscr{V}\mathtt{ar}_s$ for variables. The other kit operations are also straightforward to define: for example, mapping from variables is just the identity, and mapping to terms is the `var` constructor.

Term traversal The generic term traversal applies a substitution (for any schema $\mathscr{S}$) to every variable of a term or computation. This is the only function in our framework which is defined via structural recursion on the argument, so adding a new term to the syntax requires only one function to be modified. Below we list some of the interesting cases; the others are either similar, or the traversal simply distributes over the constructor (e.g. in pairing or injections). It is worth noting that this definition is quite clean and relies only on kit or substitution combinators.

```
traverse : ∀{𝒮 Γ Δ A} -> Subst 𝒮 Γ Δ -> Γ ⊢ A -> Δ ⊢ A
traverse σ (var x)          = 𝓉 (subst-var σ x)
traverse σ (lam M)          = lam (traverse (σ ↑ k) M)
traverse σ (M $ N)          = traverse σ M $ traverse σ N
traverse σ (persist M)      = persist (traverse (σ ↓ˢ k) M)
traverse σ (letSta S In M)  = letSta traverse σ S
                                   In traverse (σ ↑ k) M
traverse σ (event E)        = event (traverse' σ E)

traverse' : ∀{𝒮 Γ Δ A} -> Subst 𝒮 Γ Δ -> Γ ⊨ A -> Δ ⊨ A
traverse' σ (pure M)          = pure (traverse σ M)
traverse' σ (letEvt E In C) = letEvt traverse σ E
                                   In traverse' (σ ↓ˢ k ↑ k) C
```

In the `var` case, we use our `subst-var` function from above to apply the substitution to the variable, then map the resulting schema to a term with $\mathcal{t}$. In the `lam` case (and every other case with a binder involved), we recursively traverse the body of the lambda, but lift the substitution over the bound variable. Traversal distributes over application, and most other non-binding compound terms. The `persist` case makes use of our combinator $\downarrow^s$ to stabilise the context of `M`. In `event` and `pure`, we make mutually recursive calls to traverse the inner computations or terms, respectively. Finally, in event binding, we sequence the context stabilisation and lifting combinators to traverse the body.

As expected, we derive the `rename` and `substitute` functions by specialising `traverse` to variables and terms, respectively[2]. At this point, we are ready to construct the concrete version of any structural lemma or substitution we need, just by applying `substitute` (or its computational analogue `substitute'`) to an explicit substitution we defined in the previous section. For example:

[2]The implementation is a bit more complicated than that, as we need `rename` to define term weakening which is used in the declaration of the $\mathscr{T}\mathtt{erm}$ kit, and we need the $\mathscr{T}\mathtt{erm}$ kit to define `substitute` which is used in the declaration of $\mathscr{T}\mathtt{erm}_s$ – this is why we couldn't put the $\mathfrak{s}$ field in the `Kit` record directly.

```
exchange : ∀{Γ A B C} ->   Γ, A, B ⊢ C
                          -------------
                     ->   Γ, B, A ⊢ C
exchange = substitute (ex-top_s 𝒯erm_s)

[_/] : ∀{Γ A B}  -> Γ ⊢ A   ->   Γ, A  ⊢ B
                   ------------------------
                 ->         Γ ⊢ B
[M /] = substitute (sub-top_s 𝒯erm_s M)

[_/'] : ∀{Γ A B} -> Γ ⊢ A   ->   Γ, A  ⊨ B
                   ------------------------
                 ->         Γ ⊨ B
[M /'] = substitute' (sub-top_s 𝒯erm_s M)
```

Despite the relative complexity of our language, we can specify very different metatheoretic properties as instances of a single term traversal operation. Without syntactic kits and substitutions, we would need to do repetitive structural induction on terms in every definition. Instead, we abstract out the term analysis into a generic traversal function and describe the desired properties as explicit substitutions. The added benefit of this syntactic framework is that it has a semantic analogue, as demonstrated below; this allows us to prove the soundness of substitution and other lemmas in a similarly concise, high-level way.

Substitution of computations The syntactic kit framework allows us to substitute terms into other terms or computations. However, we do not yet have a way to substitute computations into computations, which is required for evaluation of event binding and selection. Unfortunately, due to the non-standard typing rules and stabilisation, we cannot provide `Kit` or `SubstKit` instances for ⊨, so our generic term traversal will not work for this case. It is not surprising that computations have to be treated separately: as Pfenning and Davies show, this operation has to be defined in a non-standard way, analysing the structure of the substituted computation, not the host.

```
⟨_/⟩ : ∀{Γ A B} -> Γ ⊨ A now  ->   Γˢ, A now ⊨ B now
                  ------------------------------------
                ->                Γ ⊨ B now
⟨ pure M         /⟩ D = substitute' (sub-top^s_s 𝒯erm_s M) D
⟨ letSig S InC C  /⟩ D =
        letSig S InC ⟨ C /⟩ (substitute' ((id_s 𝒯erm) ⁺ 𝒯erm ↑ 𝒯erm) D)
⟨ (letEvt E In C) /⟩ D =
        letEvt E In  ⟨ C /⟩ (substitute' ((Γ ^ss_s 𝒯erm) ↑ 𝒯erm) D)
```

In the `pure` case, we substitute the term `M` into the computation `D`. In the binding terms (and `select`, similar to `letEvt`) we recurse on the body of the term after lifting and weakening or stabilising its context.

Semantic kits

As shown above, syntactic kits are a collection of functions necessary to define a generic term traversal with an explicit substitution. Any context transformation lemma (e.g. substitution or weakening) can then be expressed in terms of this traversal.

Semantic kits allow us to prove that this traversal lemma is sound in our categorical semantics, and ultimately derive the soundness proof of substitution: the denotation of a term with a substitution is equal to the term interpreted in the context transformed by the substitution:

$$\llbracket [\sigma]M \rrbracket_{\mathrm{m}} = \llbracket M \rrbracket_{\mathrm{m}} \circ \llbracket \sigma \rrbracket_{\mathrm{s}}$$

A semantic kit is a collection of lemmas, one for each kit operation, along with a way to interpret the underlying schema. The lemmas are chosen as the minimal properties required to derive the soundness of traversal – they might seem oddly specific, but they are also much easier to prove than the full soundness property from first principles. We believe that this technique eliminates a lot of boilerplate that comes with substitution proofs, just like how syntactic kits simplified the definition of these operations.

Below is the record describing syntactic kits. The ⟦_⟧ function interprets the schema as a **Reactive** morphism. The laws ⟦𝓋⟧ and ⟦𝓉⟧ ensure that 𝓋 and 𝓉 work as expected: mapping the top variable to the schema is interpreted as the second projection, and the interpretation of the schema coincides with the denotation of the term created from the schema. Interpreting a weakened schema is the same as its denotation in the tail of the context. Finally, stabilising the context then interpreting 𝒶 T is more-or-less the same as interpreting T. This lemma implies the more intuitive condition $\llbracket 𝒶\ \mathtt{T} \rrbracket \circ \llbracket ^{\mathrm{s}} \rrbracket_\Gamma = \llbracket \mathtt{T} \rrbracket$.

```
record ⟦Kit⟧ {𝒮 : Schema} (k : Kit 𝒮) : Set where
  field
    ⟦_⟧  : ∀{Γ A} -> 𝒮 Γ A -> ⟦Γ⟧x ⇾ ⟦A⟧j
    ⟦𝓋⟧  : ∀{Γ A}                    -> ⟦𝓋 top⟧ = π2
    ⟦𝓉⟧  : ∀{Γ A} (T : 𝒮 Γ A)        -> ⟦𝓉 T⟧m  = ⟦T⟧
    ⟦𝓌⟧  : ∀{Γ A} (T : 𝒮 Γ A)        -> ⟦𝓌 T⟧   = ⟦T⟧ ∘ π1
    ⟦𝒶⟧  : ∀{Γ A} (T : 𝒮 Γ (A always)) -> □⟦𝒶 T⟧ ∘ ⟦ˢ⟧□Γ = δ⟦A⟧t ∘ ⟦T⟧
```

Along with the semantic kits, we also need to interpret explicit substitutions. A substitution $\Delta \vdash \sigma \triangleright \Gamma$ has the denotation $\llbracket \Delta \rrbracket_{\mathrm{x}} \rightarrowtriangle \llbracket \Gamma \rrbracket_{\mathrm{x}}$: it maps the free variables of the substituted term to the free variables of the “host”. In Agda, this can be expressed as:

```
⟦_⟧s : ∀{𝒮 Γ Δ} -> Subst 𝒮 Γ Δ -> ⟦Δ⟧x ⇾ ⟦Γ⟧x
⟦  •  ⟧s = !
⟦σ ▸ T⟧s = ⟨⟦σ⟧s, ⟦T⟧⟩
```

We can use these definitions to state soundness lemmas for the individual substitution combinators, which thus become “proof combinators”. We state the types below for reference, and one example proof (the others are similar). In all lemmas `k : Kit 𝒮` is a syntactic kit and σ : `Subst 𝒮 Γ Δ` is a substitution.

```
⟦+⟧   : ∀{Γ Δ A} -> ⟦σ + k⟧s = ⟦σ⟧s ∘ π1
⟦↑⟧   : ∀{Γ Δ A} -> ⟦σ ↑ k⟧s = ⟦σ⟧s × idA
⟦ids⟧ : ∀{Γ}     -> ⟦ids⟧s = idΓ
⟦↓s⟧  : ∀{Γ Δ A} -> □⟦σ ↓s k⟧s ∘ ⟦s⟧□Δ = ⟦s⟧□Γ ∘ ⟦σ⟧s

⟦↑⟧  : ∀{Γ Δ A} -> ⟦σ ↑ k⟧s = ⟦σ⟧s × idA
⟦↑⟧     •     rewrite ⟦𝓋⟧ = refl
⟦↑⟧ (σ ► T) rewrite ⟦+⟧ σ | ⟦𝓌⟧ T | ⟦𝓋⟧ = refl
```

Note that $[\![\downarrow^s]\!]$ is actually the naturality condition for $[\![{}^s]\!]\square$: as substitutions are morphisms in the category of contexts, applying the $[\![\_]\!]_x$ functor to $\Delta \vdash \sigma \triangleright \Gamma$ is the same as interpreting it with $[\![\_]\!]_s$.

$$\begin{tikzcd}
{[\![\Delta]\!]_x} \arrow[r, "{[\![{}^s]\!]\square_\Delta}"] \arrow[d, "{[\![\sigma]\!]_s}"'] & {\square[\![\Delta^s]\!]_x} \arrow[d, "{[\![\sigma \downarrow^s k]\!]_s}"] \\
{[\![\Gamma]\!]_x} \arrow[r, "{[\![{}^s]\!]\square_\Gamma}"'] & {\square[\![\Gamma^s]\!]_x}
\end{tikzcd}$$

We have a similar exchange condition for substitution and `handle`:

```
handle-comm : handle (⟦C1⟧c ∘ σ ↓s k ↑ k ↑ k)
                     (⟦C2⟧c ∘ σ ↓s k ↑ k ↑ k)
                     (⟦C3⟧c ∘ σ ↓s k ↑ k ↑ k)
            = handle ⟦C1⟧c ⟦C2⟧c ⟦C3⟧c ∘ ((σ ↓s k) × id⟦A⟧t⊛⟦B⟧t)
```

Soundness of traversal

Semantic kits allow us to write a general soundness proofs for term traversal and then instantiate it to various soundness lemmas including weakening and substitution. The theorem says that the denotation of a term $\Gamma \vdash M : A$ traversed with $\Delta \vdash \sigma \triangleright \Gamma$ is equal to the term interpreted in the context transformed with $[\![\sigma]\!]_s$.

$$[\![\text{traverse } \sigma\ M]\!]_m = [\![M]\!]_m \circ [\![\sigma]\!]_s$$

Again, we focus on the interesting cases.

```
traverse-sound : ∀{𝒮 Γ Δ A} (σ : Subst 𝒮 Γ Δ) (M : Γ ⊢ A)
               -> ⟦traverse σ M⟧m = ⟦M⟧m ∘ ⟦σ⟧s
traverse-sound • (var ())   -- Impossible case
traverse-sound (σ ► T) (var top) = ⟦t⟧ T
traverse-sound (σ ► T) (var (pop x)) = traverse-sound σ (var x)
traverse-sound σ (lam M) rewrite traverse-sound (σ ↑ k) M | ⟦↑⟧ σ = refl
traverse-sound σ (M $ N) rewrite traverse-sound σ M
                               | traverse-sound σ N = refl
traverse-sound σ (extract M) rewrite traverse-sound σ M = refl
```

```
traverse-sound σ (persist M) rewrite traverse-sound (σ ↓ˢ k) M
                                    | ⟦↓ˢ⟧ σ = refl
traverse-sound σ (letSta M In N) rewrite traverse-sound σ M
                                        | traverse-sound (σ ↑ k) N
                                        | ⟦↑⟧ σ = refl
traverse-sound σ (event E) = traverse'-sound σ E
```

The structure of the proof follows the definition of traversals, by induction on the term `M`. In compound terms (pairs, projection, etc.) the traversal distributes over the structure, so it is sufficient to recursively establish the soundness of the component traversals. In the `var` case, we split on the variable index; if it is the top (i.e. we found the variable we want to substitute for), we prove the equality of denotations with our $[\![\mathscr{t}]\!]$ lemma – otherwise, we recurse on the other indices. In the `lam` case, we apply the induction hypothesis proving that the traversal of the body with a lifted substitution is sound, then use $[\![\uparrow]\!]$ to show that the lifting is sound. For application we simply recurse on the subterms.

The cases for `extract` and `stable` are also recursive applications of the IH. The `persist` case is similar to `lam`, except we stabilise the substitution instead of lifting it. Signal binding is also analogous to `lam`, with an extra recursive call on the bound term. Finally, for events we call the computational version of the soundness lemma, shown below.

```
traverse'-sound : ∀{𝒮 Γ Δ A} (σ : Subst 𝒮 Γ Δ) (C : Γ ⊨ A)
              -> ⟦traverse' σ C⟧c = ⟦C⟧c ∘ ⟦σ⟧s
traverse'-sound σ (pure M) rewrite traverse-sound σ M = refl
traverse'-sound σ (letSig M In C) rewrite traverse-sound σ M
                                        | traverse'-sound (σ ↑ k) C
                                        | ⟦↑⟧ σ = refl
```

The lemmas for event binding and selection have longer, but similarly high-level proofs: the steps involve the IH, functor laws and naturality condition for $\mathrm{st}^{\square}$. As an example, the full proof for the event binding can be found in Appendix C.

Soundness of context lemmas

The soundness of traversal lemma is a general result that can be used with any instance of a semantic kit. Given the simplicity of the lemmas, it is not difficult to define semantic kits for variables ($[\![\mathscr{V}\mathtt{ar}]\!]$) and terms ($[\![\mathscr{T}\mathtt{erm}]\!]$). Then, the derived soundness of substitution proof is:

```
substitute-sound : ∀{Γ Δ A} (σ : Subst Term Γ Δ) (M : Γ ⊢ A)
                -> ⟦substitute σ M⟧m = ⟦M⟧m ∘ ⟦σ⟧s
substitute-sound σ M = traverse-sound ⟦𝒯erm⟧ σ M
```

Given any substitution σ, we can derive a soundness proof. The specific substitution we need (used in the equational theory) is replacing the top variable of the context with a term `M`, which we defined in Appendix B as $\mathtt{sub\text{-}top}_s$ $[\![\mathscr{T}\mathtt{erm}]\!]$ `M`. In fact, we can provide a full categorical proof by establishing that the denotation of this substitution is equal to $\langle \mathtt{id}_\Gamma, [\![\mathtt{M}]\!]_m \rangle$:

```
⟦sub-topₛ⟧ : ∀{Γ A} -> (M : Γ ⊢ A) -> ⟦sub-topₛ ⟦𝒯erm⟧ M⟧ₛ = ⟨idΓ, ⟦M⟧ₘ⟩
⟦sub-topₛ⟧ M rewrite ⟦idₛ⟧ₛ = refl

subst-sound : ∀{Γ A B} (M : Γ ⊢ A) (N : Γ, A ⊢ B)
           -> ⟦[M /] N⟧ₘ = ⟨idΓ, ⟦M⟧ₘ⟩
subst-sound M N rewrite sym (⟦sub-topₛ⟧ M) =
    substitute-sound (sub-topₛ 𝒯ermₛ M) N
```

This is the standard expression for soundness of substitution in a CCC, which demonstrates that despite their overall simplicity, syntactic and semantic kits are a very adequate tool for defining syntactic operations and proving the soundness of their denotational semantics.

APPENDIX C

Proofs

Monad laws for $\Diamond$

Theorem. *$\Diamond$, together with η and μ satisfies the monad laws.*

Proof.

Lemma (Left unit law). $\mu_A \circ \eta_{\Diamond A} = \mathrm{id}_A$

Proof. The law holds by the definition of μ and η. ■

Lemma (Right unit law). $\mu_A \circ \Diamond\eta_A = \mathrm{id}_A$

Proof. In **Reactive**, this equation expands to the following, for $k : \mathbb{N}$ and $a : \bullet^k A|_n$:

$$\mu_A|_n\ (\Diamond\eta_A|_n\ (k, a)) = (k, a)$$

We now consider two cases:

Case $k \leq n$:

$$\begin{aligned}
\mu_A|_n\ (\Diamond\eta_A|_n\ (k, a)) &= \mu_A|_n\ (k, \bullet^k\eta_A|_n\ (a)) && (\Diamond \text{ map on morphisms})\\
&= (\triangleright^k_{n-k})_A(\bullet^k\eta_A|_n\ (a) \parallel \Diamond A|_{n-k}) && (\text{def. of } \mu \text{ (case } k \leq n))\\
&= (\triangleright^k_{n-k})_A(\eta_A|_{n-k}\ (a \parallel A|_{n-k})) && (\text{delay lemma})\\
&= (\triangleright^k_{n-k})_A(0, a \parallel A|_{n-k}) && (\text{def. of } \eta)\\
&= (k, a \parallel\ \bullet^k\ A|_n) && (\text{def. of } \triangleright^k_{n-k})\\
&= (k, a)
\end{aligned}$$

Case $k > n$: As $a : (\bullet^k A)_n$ and $k > n$, $a = *$.

$$\begin{aligned}
\mu_A|_n\ (\Diamond\eta_A|_n\ (k, *)) &= \mu_A|_n\ (k, \bullet^k\eta_A|_n\ (*)) && (\Diamond \text{ map on morphisms})\\
&= (k, * \parallel\ \bullet^k\ A|_n) && (\text{def. of } \mu \text{ (case } k > n))\\
&= (k, *)
\end{aligned}$$

■

Lemma (Associativity law). $\mu_A \circ \mu_{\Diamond A} = \mu_A \circ \Diamond\mu_A$

Proof. In **Reactive**, this equation expands to the following, for $k : \mathbb{N}$ and $a : (\bullet^k \Diamond\Diamond A)_n$:

$$\mu_A|_n\ (\mu_{\Diamond A}|_n\ (k, a)) = \mu_A|_n\ (\Diamond \mu_A|_n\ (k, a))$$

We now consider two cases:

Case $k \leq n$:

$$\begin{aligned}
\mu_A|_n\ (\mu_{\Diamond A}|_n\ (k, a)) &= \mu_A|_n\ ((\triangleright^k_{n-k})_{\Diamond A}(a \parallel \Diamond\Diamond A|_{n-k})) && \text{(def. of } \mu \text{ (case } k \leq n))\\
&= (\triangleright^k_{n-k})_A(\mu_A|_{n-k}\ (a \parallel \Diamond\Diamond A|_{n-k})) && \text{(interchange of } \mu \text{ and } \triangleright^k_{n-k})\\
&= (\triangleright^k_{n-k})_A(\bullet^k \mu_A|_n\ (a) \parallel \Diamond A|_{n-k})) && \text{(delay lemma)}\\
&= \mu_A|_n\ (k, \bullet^k \mu_A|_n\ (a)) && \text{(def. of } \triangleright^k_{n-k})\\
&= \mu_A|_n\ (\Diamond \mu_A|_n\ (k, a)) && (\Diamond \text{ map on morphisms})
\end{aligned}$$

Case $k > n$: As $a : \bullet^k \Diamond\Diamond A|_n$ and $k > n$, $a = *$.

$$\begin{aligned}
\mu_A|_n\ (\mu_{\Diamond A}|_n\ (k, *)) &= \mu_A|_n\ (k, * \parallel (\bullet^k \Diamond A)_n) && \text{(def. of } \mu \text{ (case } k > n))\\
&= (k, * \parallel \bullet^k\ \Diamond A|_n\ \parallel \bullet^k\ A|_n) && \text{(def. of } \mu \text{ (case } k > n))\\
&= (k, *) && \blacksquare
\end{aligned}$$

The monad laws hold, therefore $\Diamond$ is a monad.

□

Soundness of traversal for event binding

Theorem. *Traversal of an event binding computation is sound: for all contexts* Γ *and* Δ*, substitutions* $\Delta \vdash \sigma \triangleright \Gamma$*, terms* $\Gamma \vdash E : \mathsf{Event}\ A\ \mathsf{now}$ *and computations* $\Gamma^{\mathrm{s}}, x : A\ \mathsf{now} \vdash C \div B\ \mathsf{now}$*, we have:*

$$[\![\mathsf{traverse}'\ \sigma\ (\textbf{let evt}\ x = E\ \textbf{in}\ C)]\!]_{\mathrm{c}} = [\![\textbf{let evt}\ x = E\ \textbf{in}\ C]\!]_{\mathrm{c}} \circ [\![\sigma]\!]_{\mathrm{s}}$$

Proof. Assume the following induction hypotheses:

$$[\![\mathsf{traverse}\ \sigma\ E]\!]_{\mathrm{m}} = [\![E]\!]_{\mathrm{m}} \circ [\![\sigma]\!]_{\mathrm{s}}$$

$$[\![\mathsf{traverse}'\ (\sigma \downarrow^{\mathrm{s}} k \uparrow k)\ C]\!]_{\mathrm{c}} = [\![C]\!]_{\mathrm{c}} \circ [\![\sigma \downarrow^{\mathrm{s}} k \uparrow k]\!]_{\mathrm{s}}$$

We have the following equational proof:

$$\begin{aligned}
&[\![\mathsf{traverse}'\ \sigma\ (\textbf{let evt}\ x = E\ \textbf{in}\ C)]\!]_{\mathrm{c}}\\
=\ &[\![\textbf{let evt}\ x = \mathsf{traverse}\ \sigma\ E\ \textbf{in}\ \mathsf{traverse}'\ (\sigma \downarrow^{\mathrm{s}} k \uparrow k)\ C]\!]_{\mathrm{c}} && \text{(traversal of events)}\\
=\ &\mathsf{bindEvent}\ \Delta\ [\![\mathsf{traverse}\ \sigma\ E]\!]_{\mathrm{m}}\ [\![\mathsf{traverse}'\ (\sigma \downarrow^{\mathrm{s}} k \uparrow k)\ C]\!]_{\mathrm{c}} && \text{(denotation of event binding)}\\
=\ &\mathsf{bindEvent}\ \Delta\ ([\![E]\!]_{\mathrm{m}} \circ [\![\sigma]\!]_{\mathrm{s}})\ ([\![C]\!]_{\mathrm{c}} \circ [\![\sigma \downarrow^{\mathrm{s}} k \uparrow k]\!]_{\mathrm{s}}) && \text{(induction hypotheses)}\\
=\ &\mu_{[\![B]\!]_{\mathrm{t}}} \circ \Diamond([\![C]\!]_{\mathrm{c}} \circ [\![\sigma \downarrow^{\mathrm{s}} k \uparrow k]\!]_{\mathrm{s}} \circ \varepsilon_{[\![\Delta^{\mathrm{s}}]\!]_{\mathrm{x}}} \times \mathrm{id}_{[\![A]\!]_{\mathrm{t}}}) \circ \mathrm{st}^{\Box}_{[\![\Delta^{\mathrm{s}}]\!]_{\mathrm{x}}, [\![A]\!]_{\mathrm{t}}} \circ \langle [\![^{\mathrm{s}}]\!]^{\Box}_{\Delta}, [\![E]\!]_{\mathrm{m}} \circ [\![\sigma]\!]_{\mathrm{s}} \rangle\\
& && \text{(definition of bindEvent)}\\
=\ &\mu_{[\![B]\!]_{\mathrm{t}}} \circ \underline{\Diamond[\![C]\!]_{\mathrm{c}} \circ \Diamond([\![\sigma \downarrow^{\mathrm{s}} k \uparrow k]\!]_{\mathrm{s}} \circ \varepsilon_{[\![\Delta^{\mathrm{s}}]\!]_{\mathrm{x}}} \times \mathrm{id}_{[\![A]\!]_{\mathrm{t}}})} \circ \mathrm{st}^{\Box}_{[\![\Delta^{\mathrm{s}}]\!]_{\mathrm{x}}, [\![A]\!]_{\mathrm{t}}} \circ \langle [\![^{\mathrm{s}}]\!]^{\Box}_{\Delta}, [\![E]\!]_{\mathrm{m}} \circ [\![\sigma]\!]_{\mathrm{s}} \rangle\\
& && (\Diamond \text{ functor law})
\end{aligned}$$

$$\begin{aligned}
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond\llbracket C\rrbracket_{\mathrm{c}} \circ \Diamond(\underline{\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}} \circ \varepsilon_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \mathrm{st}^{\square}_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill \text{(denotation of } \uparrow) \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond\llbracket C\rrbracket_{\mathrm{c}} \circ \Diamond(\underline{(\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}} \circ \varepsilon_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}}) \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}}) \circ \mathrm{st}^{\square}_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill \text{(CCC law)} \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond\llbracket C\rrbracket_{\mathrm{c}} \circ \Diamond(\underline{(\varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \circ \square\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}})} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \mathrm{st}^{\square}_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill \text{(naturality of } \varepsilon) \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond\llbracket C\rrbracket_{\mathrm{c}} \circ \Diamond(\underline{\varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}} \circ \square\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}}) \circ \mathrm{st}^{\square}_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill \text{(CCC law)} \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond\llbracket C\rrbracket_{\mathrm{c}} \circ \underline{\Diamond(\varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \Diamond(\square\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}})} \circ \mathrm{st}^{\square}_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill (\Diamond \text{ functor law}) \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \underline{\Diamond(\llbracket C\rrbracket_{\mathrm{c}} \circ \varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}})} \circ \Diamond(\square\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \mathrm{st}^{\square}_{\llbracket \Delta\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill (\Diamond \text{ functor law}) \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond(\llbracket C\rrbracket_{\mathrm{c}} \circ \varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \underline{\mathrm{st}^{\square}_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \square\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}} \circ \langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill \text{(naturality of } \mathrm{st}^{\square}) \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond(\llbracket C\rrbracket_{\mathrm{c}} \circ \varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \mathrm{st}^{\square}_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \underline{\langle \square\llbracket \sigma \downarrow^{\mathrm{s}} k\rrbracket_{\mathrm{s}} \circ \llbracket^{\mathrm{s}}\rrbracket\square_{\Delta}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle} \\
&\hfill \text{(CCC law)} \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond(\llbracket C\rrbracket_{\mathrm{c}} \circ \varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \mathrm{st}^{\square}_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \langle \underline{\llbracket^{\mathrm{s}}\rrbracket\square_{\Gamma} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}}, \llbracket E\rrbracket_{\mathrm{m}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}}\rangle \\
&\hfill \text{(denotation of } \downarrow^{\mathrm{s}}) \\
&= \mu_{\llbracket B\rrbracket_{\mathrm{t}}} \circ \Diamond(\llbracket C\rrbracket_{\mathrm{c}} \circ \varepsilon_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}}} \times \mathrm{id}_{\llbracket A\rrbracket_{\mathrm{t}}}) \circ \mathrm{st}^{\square}_{\llbracket \Gamma\,^{\mathrm{s}}\rrbracket_{\mathrm{x}},\llbracket A\rrbracket_{\mathrm{t}}} \circ \underline{\langle \llbracket^{\mathrm{s}}\rrbracket\square_{\Gamma}, \llbracket E\rrbracket_{\mathrm{m}}\rangle \circ \llbracket \sigma\rrbracket_{\mathrm{s}}} && \text{(CCC law)} \\
&= \underline{\mathsf{bindEvent}\ \Gamma\ \llbracket E\rrbracket_{\mathrm{m}}\ \llbracket C\rrbracket_{\mathrm{c}}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}} && \text{(definition of } \mathsf{bindEvent}) \\
&= \llbracket \mathbf{let\ evt}\ x = E\ \mathbf{in}\ C\rrbracket_{\mathrm{c}} \circ \llbracket \sigma\rrbracket_{\mathrm{s}} && \text{(denotation of event binding)}
\end{aligned}$$

□