Programming language semantics in modal type theories

Philipp Stassen

PhD Dissertation

Department of Computer Science
Aarhus University
Denmark

Programming language semantics in modal type theories

A Dissertation
Presented to the Faculty of Natural Sciences
of Aarhus University
in Partial Fulfillment of the Requirements
for the PhD Degree

by
Philipp Stassen
May 29, 2024

Abstract

Modalities provide powerful abstractions for various problems in mathematics and computer science. For example, they find applications in staged programming [44], program security properties [1, 70], reactive programming [15, 71], concurrency [87] and step-indexing [22, 89].

Type theory combines logic and programming in one language. This allows us to write and reason about programs in type theory, as well as to interpret other programming languages.

The goal of this thesis is twofold: On one hand, we explore applications of a specific modal type theory, namely guarded type theory, in programming language semantics. On the other hand, we describe how a general framework for modal type theories can be turned into a proof assistant.

In §2 and §3 we show how to use a constructive guarded type theory to reason about FPC, a programming language with recursive types, and $\mathsf{FPC}_{\oplus}$, which extends FPC by probabilistic choice. We define both operational and denotational semantics for FPC and $\mathsf{FPC}_{\oplus}$ and prove soundness. By doing this, we provide the first semantics of $\mathsf{FPC}_{\oplus}$ in constructive type theory. Finally, we construct a relation between syntax and semantics and show how to prove contextual equivalence between programs. It is difficult to reason about languages with recursion and computational effects such as probabilistic choice, because these features have no obvious counterpart in type theory. By using the later modality of guarded type theory and a special guarded fixpoint combinator, it is possible to define semantic domains that interpret these effects.

Using modalities to abstract low-level details away significantly simplifies the task at hand for practitioners. To truly benefit from these capabilities, a proof assistant that automates simpler arguments and validates the correctness of complicated proofs is of the essence. However, implementing a proof assistant is a difficult undertaking and unfortunately, many modalities cannot be handled by current proof assistants.

In §4 we explain how to implement the general modal type theory MTT, which can internalize many different modal situations. Importantly, this includes a guarded type theory comparable to the one used in §2 and §3. However, the framework is general enough to internalize arbitrary collections of (dependent) right adjoints [25]. The resulting proof assistant `mitten` is modular and specializes to many modal situations.

Resumé

Modal operatorer er blevet studeret i mange år i matematisk logic. Modal operatorer finder også anvendelse inden for datalogi, hvor de for eksempel bruges ifbm. partiel evaluering [44], type sysetmer for non-interference [1, 70], reaktiv programmering [15, 71], concurrency [87] og såkaldte step-indekserede programlogikker [22, 89].

Typeteori kombinerer logik og programmering i ét sprog. Det er nyttigt både til at ræsonnere over programmer skrevet i typeteori og til at ræsonnere over andre programmeringssprog inden for typeteori.

Målet med denne afhandling er todelt: På den ene side udforsker vi anvendelser af modale typeteorier i programmeringssprogs semantik. På den anden side beskriver vi, hvordan en generel teori for modale typeteorier kan implementeres i en bevisassistent.

I §2 og §3 viser vi hvordan man bruger guarded typeteori til at definere og ræsonnere om FPC, et programmeringssprog med rekursive typer, og $\mathsf{FPC}_{\oplus}$, som udvider FPC med probabilistiske valg. Vi definerer både operationel og denotationel semantik og beviser sundhed. Endelig konstruerer vi en relation mellem de to og viser, hvordan man kan bruge den til at ræsonnere om programmer op til kontekstuel ækvivalens.

Det er velkendt, at det er meget udfordrende at ræsonnere om sprog med rekursion og beregningseffekter som f.eks. probabilistiske valg, fordi disse effekter er svære at repræsentere i typeteori. Ved at anvende guarded typeteori er det muligt at definere semantiske domæner, der fortolker disse effekter.

En af fordelene ved modal typeteori er at man kan abstrahere fra mange lavniveau detaljer. For virkelig at drage fordel af disse muligheder er det vigtigt med en bevisassistent, der automatiserer enklere argumenter og validerer korrektheden af komplicerede beviser. Det er dog en vanskelig opgave at implementere en bevisassistent, og desværre kan mange modaliteter ikke håndteres af de eksisterende bevisassistener.

I §4 viser vi, hvordan man implementerer den generelle modale typeteori MTT, som kan internalisere mange forskellige modale situationer. MTT inkluderer en variant af guarded typeteori, lig den der bruges i §2 og §3. MTT er generel nok til at internalisere vilkårlige samlinger af (afhængige) højre adjoints [25]. Den resulterende bevisassistent `mitten` er modulær og kan instantieres til mange specifikke modale situationer.

Acknowledgments

I am indebted to my supervisor Lars Birkedal, without whose guidance this thesis would have never materialized. It is the culmination of a decade-long academic and personal journey, and for either, Lars generously supported me in whatever path I chose. His strategic foresight helped me to select fruitful research projects and was invaluable for my scientific success. For all of this, I am truly grateful. Also, my thanks go to the Villum Fonden, who generously financed my PhD.

I am very grateful to Robert Atkey and Neel Krishnaswami who accepted to be on my PhD committee. It is a great honor to have two such renowned and successful experts in the field evaluating my research.

I furthermore am very thankful for my colleagues and collaborators, of whom I would like to highlight Alejandro Aguirre, Gustavo Bertoli, Luca Castilgione, Daniel Gratzer, Emil Lupo, Rasmus Møgelberg, Daniel Pereira, Cora Perner, and Maaike Zwart. Here, I want to emphasize Daniel's and Rasmus' contributions to my research, who at times were more like mentors to me. Finally, I thank Lars Birkedal, Jonas Kastberg Hinrichsen, Magnus Kristensen and Rasmus Møgelberg for proofreading and comments.

I greatly benefited from the fantastic research environment of the Aarhus logic and semantics group. My thanks go to all the people who contributed to this inspiring atmosphere. I furthermore thank the Airbus Cyber Security department and my colleagues and collaborators from the Albatross and Concordia Project for making me part of the team. I felt honored and appreciated by your trust.

More personally, I thank my loving parents Anna and Bernd, who are the bedrock of my academic career. From the very beginning, they wholeheartedly supported all of my endeavors unconditionally, and I hope this thesis honors all your efforts. Then, I am fortunate to have my little sister Nina, who has grown to be a close friend. I am most grateful to my partner Sarah Fassoth, for her love and care. You made the second leg of my PhD the happiest time. Furthermore, I truly appreciate my friend Elvin Ruic, who kept me sane during the Covid pandemic and ever after. Similarly, I'd like to acknowledge my friends from philosophy in the mountains, who cultivated my passion for formal logic.

It is the worst-kept secret that winter bathing was my way of coping with the Danish winter and I was lucky to have in Simon Friis Vindum somebody to share this approach. Finally, my years in Aarhus would have been far less enjoyable without my colleagues and friends Alejandro Aguirre, Philipp Haselwarter, Jonas Kastberg

Hinrichsen and Magnus Kristensen.

I am blessed with a large extended family as well as many other pivotal friends in my life and I deeply appreciate all of you.

At last, I remain in love with our 14.5-year-old family dog Emma, who was diagnosed with terminal cancer in the last days of writing this thesis. While your manners are at best average, your appetite surely is not. You are the finest and kindest dog, and if there were any flaws they can with certainty be attributed to a bad upbringing (we did our best). I hope you enjoy the final leg of your journey!

Philipp Stassen,
Aarhus, May 29, 2024.

Contents

Abstract i

Resumé iii

Acknowledgments v

Contents vii

I Overview 1

1 Introduction 3
- 1.1 Coinduction and partiality . . . 5
- 1.2 Guarded Type Theory . . . 8
- 1.3 Clocked Cubical Type Theory . . . 11
- 1.4 Implementing a flexible multimodal proof assistant . . . 25
- 1.5 Contributions and Structure . . . 29

II Publications 33

2 Technical Report: Modeling FPC in Guarded Type Theory 35
- 2.1 Introduction . . . 35
- 2.2 The programming language FPC . . . 36
- 2.3 Denotational Semantics . . . 49
- 2.4 Lifting Relations . . . 61
- 2.5 A logical relation for contextual refinement . . . 63
- 2.6 Examples . . . 71
- 2.7 Related works . . . 76
- 2.8 Conclusion . . . 77

3 Modelling Probabilistic FPC in Guarded Type Theory 79
- 3.1 Introduction . . . 79
- 3.2 Clocked Cubical Type Theory . . . 81

3.3 Finite distributions . . . 84
3.4 Convex delay algebras . . . 88
3.5 Probabilistic FPC . . . 93
3.6 Denotational semantics . . . 96
3.7 Couplings and Lifting relations . . . 98
3.8 Relating syntax and semantics . . . 102
3.9 Examples . . . 103
3.10 Related Work . . . 106
3.11 Conclusion and Future Work . . . 107

4 `mitten` : a flexible multimodal proof assistant 109
4.1 Introduction . . . 109
4.2 A surface syntax for MTT . . . 113
4.3 Normalization by Evaluation . . . 118
4.4 Implementing a Mode Theory . . . 122
4.5 Semantic Type-Checking Algorithm . . . 124
4.6 Case study: guarded recursion in `mitten` . . . 127
4.7 Related Work . . . 131
4.8 Conclusions and future work . . . 132

Appendix 135
Omitted proofs of chapter 3 . . . 137

Bibliography 167

Part I

Overview

Chapter 1

Introduction

Modal type theories have been used widely to abstract concrete problems in computer science and mathematics. To name a few examples, the different iterations of guarded type theories use the later modality to integrate guarded recursion into type theory [14, 23, 25, 27, 57, 60, 72, 89]. Additionally, one can realize coinductive types by combining guarded recursion with the everything now modality [28, 36] or clock quantification [13, 72, 80].

Cohesive homotopy type theory [99] uses modalities to combine the viewpoints of topology and homotopy theory in one framework (as proposed by Lawvere [74]).

More on the computer science side of things, Kavvos [70] uses the modalities of [99] to tackle program security properties, refining previous work of Abadi et al. [1]. Furthermore, modalities have been used for distributed systems [87], reactive programming [15, 71] and staged computation [44]. They also proved to be useful in higher-order concurrent separation logics [23, 69].

The goal of this thesis is twofold: On one hand we explore applications of guarded type theories in programming language semantics. On the other hand, we describe how a general framework for modal type theories can be turned into a proof assistant.

Programming language semantics in guarded type theory Intensional type theories recently got more appreciation, as the emergence of homotopy type theory [106] displayed that type theories can be very useful reasoning tools if they are related to ordinary mathematics (as HoTT is via the simplicial sets interpretation). Homotopy Type Theory simplifies many difficult constructions of homotopy theory and most importantly allows computer-aided verification of complicated proofs.

In the same spirit, guarded type theories are a powerful abstraction of step-indexing techniques [11, 12] and metric domain theory [10, 49]. The archetypical example of a guarded type theory is the internal language of the topos of trees [23].

Our goal is to use this connection between type theories and mathematics to interpret effectful programming languages. More specifically, we investigate how one can develop operational and denotational semantics of FPC — a programming language with recursive types — as well as its probabilistic extension $\mathsf{FPC}_{\oplus}$ in a

constructive guarded type theory. Since $\mathsf{FPC}_{\oplus}$ supports the sampling of random values from a probability distribution, a program evaluates to distribution over values, as opposed to a single value in the case of ordinary deterministic computation. In combination with general recursion, this allows us to write programs that evaluate to distributions with infinite support.

Interpreting a higher-order probabilistic programming language with recursion is no easy task, and specifically, two factors complicate such a development for us:

1. We want to interpret recursive types $\mu X.\tau$ and thus there ought to be a domain validating $[\![\mu X.\tau]\!] \triangleq [\![\tau[\mu X.\tau/X]]\!]$ definable in our meta theory. Unlike with e.g. products or coproducts, there is no meta-level type former that could realize such an interpretation directly: It is well known that type theories cannot have unrestricted recursion, as otherwise the relation to ordinary math is lost.

   Phrased differently, FPC and $\mathsf{FPC}_{\oplus}$ encompass non-terminating programs, which do not have direct counterparts in type theory.

2. In previous works, the operational semantics of $\mathsf{FPC}_{\oplus}$ relies on real numbers and non-constructive reasoning. This is not possible in a constructive meta-theory.

It turns out that both of these challenges can be met by combining features of homotopy type theory and guarded type theory.

The specific type theory we use is Clocked Cubical Type Theory (CCTT), which combines *cubical type theory* (a variant of homotopy type theory) with *clocked type theory*. It provides the modal type former $\triangleright^{\kappa}$ (pronounced 'later'), which is indexed by a clock κ. Terms of type $\triangleright^{\kappa}A$ can be thought of only being accessible after one computation step. We can now use the guarded fixpoint combinator fix^{κ} to define the guarded partiality monad[1] L^{κ} which validates $\mathsf{L}^{\kappa}A \simeq A + \triangleright^{\kappa}(\mathsf{L}^{\kappa}A)$. This can be used to interpret recursive types.

To address the second challenge, we marry a distribution monad with the guarded partiality monad, we call this novel monad *guarded convex delay monad* and denote it by D^{κ}. It is defined by $\mathsf{D}^{\kappa}A \simeq \mathscr{D}(A + \triangleright^{\kappa}(\mathsf{D}^{\kappa}A))$, where the finite distibution monad $\mathscr{D}$ can be constructed as a *higher inductive type*.

Using this, we can define the operational and denotational semantics of $\mathsf{FPC}_{\oplus}$ completely in CCTT. We furthermore internalize the concept of *contextual equivalence* — the correct notion of equality between terms of a programming language. While the denotational semantics does not respect contextual equivalence, we can use guarded recursion to define a relation between semantics and syntax which implies it. This relies yet on another feature of CCTT, namely *clock quantification*, which collapses the delays induced by $\triangleright^{\kappa}$ [13]. We have that $\forall\kappa.\triangleright^{\kappa}A \simeq \forall\kappa.A$, which we crucially leverage when proving that our logical relation (which was defined by guarded recursion) implies the (*not step-indexed*) notion of contextual refinement. In the end, we display the utility of such an approach with various examples.

[1] This monad is also known as 'guarded delay monad'

This dissertation contains a technical report §2 which showcases this technique on call-by-value FPC. §3 is the paper expanding these results to $\mathsf{FPC}_{\oplus}$.

Implementing a multimodal proof assistant Using modalities to abstract low-level details away significantly simplifies the task at hand for practitioners. To truly benefit from these capabilities, a proof assistant that automates simpler arguments and validates the correctness of complicated proofs is of the essence. However, implementing proof assistants is a difficult undertaking and unfortunately, many modalities cannot be handled by the existing ones.

It is in general challenging to integrate modalities into dependent type theories. This is because they often do not fit into their framework — in particular, they do not necessarily respect substitution. Therefore, to obtain a workable type theory, one has to augment the judgmental structure, using dual context calculi [92] or a Fitch-style type theory to finesse around the problem. Since such a structure is unique to the modal situation, every modality requires its own proof assistant — an unreasonable effort.

Birkedal et al. [24] identified a class of modalities, namely those that behave like *dependent right adjoints*, which can be treated in a unified framework by using a Fitch-style type theory. This approach has been refined by Gratzer et al. [58] who introduced multimode type theory (MTT), a framework, which can be instantiated with arbitrary collections of dependent right adjoints. In particular, it allows combinations of different modalities and to define interactions between them. Importantly, the everything now modality [28, 36] and guarded recursion fall in this framework. This allows us to realize a type theory with coinductive types and is thereby an alternative approach to clocked type theory.

A proof assistant for MTT should be modular and specialize to any modal situation — thereby realizing the proof assistant that was previously impractical.

In the following, we start by informally introducing high-level ideas underlying coinductive types (1.1) and guarded type theory (1.2). Then, in Section 1.3 we give an account of *clocked cubical type theory* and show how it realizes these concepts. Additionally, we

In Section 1.4, we introduce MTT as well as the relevant algorithms that are needed to implement it. This in particular includes normalization-by-evaluation and bidirectional type checking.

1.1 Coinduction and partiality

It is well-known that general recursion breaks the Curry-Howard isomorphism, and thereby well-typed terms do not correspond to proofs anymore. The problem is that functions in type theories are total, but general recursion allows us to define partial functions.

In his work Moggi [86] established that effectful computations can be naturally interpreted in specific monads. A partial function from A to B is thereby viewed as a

total function of type $A \to \mathscr{M}(B)$, where $\mathscr{M}(X) \triangleq X + \bot$. Thus, $\mathscr{M}$ is often referred to as the *partiality monad*. When interpreting programming languages, $\bot$ represents the non-terminating program.

However, in a constructive setting, this is insufficient since we are not able to decide whether a program terminates or not. As an alternative, Capretta [29] proposed to use a coinductive partiality monad instead.

In the following, we introduce the concepts of induction and coinduction by investigating the examples of *lists* (as an inductive type) and *streams* (as a coinductive type). In its purest form, both induction and coinduction stem from the study of algebras in mathematics [65].

Induction at the example of lists We denote the collection of *lists* of type A by list_A. From a computer science perspective, list_A should be freely generated by the constructors nil and $\mathsf{cons}(h,t)$.

In mathematics, this idea is captured by requiring that list_A is an algebra of the endofunctor $F_{\mathsf{list}_A}(X) = 1 + (A \times X)$. By the definition of F_{list_A}-algebras, this means precisely that there are maps

$$\mathsf{nil} : 1 \to \mathsf{list}_A \qquad\qquad \mathsf{cons}(h,t) : A \times \mathsf{list}_A \to \mathsf{list}_A.$$

The freeness of list_A implies also that it has the familiar induction principle, which allows us to define functions from list_A to other F_{list_A}-algebras by recursion, as long as the recursive call is nested under a constructor.

This follows by requiring that list_A is the *initial* $F_{\mathsf{list}_A}(X)$-algebra. Indeed, *induction* is the universal property of initial algebras, which gives us a unique map to any other F_{list_A}-algebra.

$$\mathsf{list}_A \dashrightarrow^{\exists!} \left(\textit{any } F_{\mathsf{list}_A}\text{-algebra}\right)$$

Figure 1.1: Induction is the universal property of initial algebras

It follows from initiality that $F_{\mathsf{list}_A}(\mathsf{list}_A) \simeq \mathsf{list}_A$ and thus $\mathsf{list_A}$ is a fixpoint (up to isomorphism) of F_{list_A}. In general, the initial algebra of an endofunctor F — if it exists — is the least fixpoint of F.

Coinduction at the example of streams On the other hand, stream_A denotes the set of streams of type A, which are best thought of as lists of infinite length. While every finite list is the result of a finite concatenation of constructors, this is not sensible for infinite lists. Therefore, dually to initial algebras, the primitive operation of streams are not constructors, but *observations*: Given a stream s, we can destruct it to its head $\mathsf{hd}(s) : A$ and tail $\mathsf{tl}(s) : \mathsf{stream}_A$.

Mathematically, this is captured by requiring that stream_A ought to be a *co-algebra* of the functor $F_{\mathsf{stream}_A}(X) \triangleq A \times X$, which is precisely a map

$$(\mathsf{hd}, \mathsf{tl}) : \mathsf{stream}_A \to F_{\mathsf{stream}_A}(\mathsf{stream}_A). \tag{1.1}$$

Being able to observe but not to construct begs the question of how we got our hands on a stream in the first place. Again, we get a satisfying mathematical answer that emphasizes the duality with induction: We require stream_A to be the *final* coalgebra, which gives us a way to synthesize streams by the *coinduction* principle. Coinduction is the universal property of final co-algebras, which gives us a unique map from any other F_{stream_A}-co-algebras into stream_A.

$$\left(\textit{any } F_{\mathsf{stream}_A}\text{-co-algebra}\right) \dashrightarrow^{\exists!} \mathsf{stream}_A$$

Figure 1.2: Coinduction is the universal property of final co-algebras

Example 1 (Synthesizing elements of $\mathsf{stream}_{\mathbb{N}}$). It follows from the finality of $(\mathsf{hd},\mathsf{tl}) : \mathsf{stream}_{\mathbb{N}} \to \mathbb{N} \times \mathsf{stream}_{\mathbb{N}}$ that there exists the morphism 'zeros' of $F_{\mathsf{stream}_{\mathbb{N}}}$-co-algebras. Consequently, zeros is an element of $\mathsf{stream}_{\mathbb{N}}$.

$$\begin{array}{ccc} 1 & \xrightarrow{\mathsf{zeros}} & \mathsf{stream}_{\mathbb{N}} \\ {\scriptstyle \lambda x.(0,x)}\downarrow & & \downarrow{\scriptstyle (hd,tl)} \\ \mathbb{N} \times 1 & \xrightarrow{\mathsf{id}\times\mathsf{zeros}} & \mathbb{N} \times \mathsf{stream}_{\mathbb{N}} \end{array}$$

Analogously to the case of initiality, finality implies that stream_A is the largest fixpoint of F_{stream_A} and in the literature this fixpoint is often named $\nu(F_{\mathsf{stream}_A})$.

Note that some elements of coinductive types can be finitely generated. For instance, the set of *co-lists* list^{∞}_A is the final co-algebra of F_{list_A} in the category of sets. Being the largest fixpoint, list^{∞}_A contains lists of all lengths, the finite and the infinite.

Example 2 (Capretta's Partiality Monad). We associate to every type A the endofunctor $F_{\mathsf{L}}(X) = A + X$. The final co-algebra $\mathsf{L}(A)$ of F_{L} validates $\mathsf{L}(A) \simeq A + \mathsf{L}(A)$ with maps

$$\eta \triangleq \mathsf{inl} - : A \to A + \mathsf{L}(A)$$
$$\mathsf{step} \triangleq \mathsf{inr} - : \mathsf{L}(A) \to A + \mathsf{L}(A)$$

Furthermore, for any type A we can define a multiplication operation $\mu_A : \mathsf{L}(\mathsf{L}(A)) \to \mathsf{L}(A)$ which respects the monad equalities. Intuitively, step represents (potentially unproductive) computation steps of programs, which — if they terminate — compute to an element of type A. Using coinduction, we can for instance define the infinite, unproductive loop $\bot : \mathsf{L}(A)$ for any type A. Capretta [29] proves that this monad is sufficient to interpret recursive functions.

Over the years different variants of types emerged that internalize the concept of induction [2, 8, 47, 53, 79]. The challenge is to capture the expressive strength of inductive types while maintaining good meta-theoretic properties — such as canonicity or normalization — and being convenient to work with. Proof assistants such as Coq and Agda therefore allow recursive definitions by pattern matching, where the programmer can define recursive functions by structural recursion on inductive data

types. This corresponds to the usage of fixpoints in functional programming languages, and a side effect of this approach is that well-typed terms do not necessarily terminate. Thus, one has to use a termination checker to ensure that the type theory is sound [38, 39, 53].

Coinductive types on the other hand can be specified by constructors or as a record type. Using the coinduction scheme requires extra *productivity* checks to guarantee that always new information is produced in finite time (note that the stronger termination requirement is not sensible for coinductive types) [4, 6, 30].

While this way of declaring coinductive types and functions is very convenient, there are two downsides for us:

- A priori endofunctors are not guaranteed to have any fixpoints at all. For example, $F(X) = X \to X$ does not have a (non-trivial) fixpoint in the category of sets. Schemes for inductive definitions therefore restrict the class of endofunctors to ensure that variables occur only strictly positively in definitions (see Pfenning and Paulin-Mohring [93] for a definition of positivity). While this excludes the aforementioned counterexample, there are also many interesting domain equations, which do not fall in this class (such as the monad modeling higher-order store for example).

- Syntactic productivity checks are not compositional and many productive functions cannot be identified. Furthermore, it is difficult to handle higher-order definitions such as $gh = 0 :: h(gh)$. The productivity of g depends on the behavior of h and therefore no verdict can be given without inspecting an explicit definition for h first.

These issues are addressed by type-based approaches such as guarded type theories or sized types [4]. In what follows we thus use a *guarded type theory*, which in particular ensures the productivity of terms via typing.

1.2 Guarded Type Theory

Nakano [89] proposed a simple type theory with the "later" modality $\triangleright$ to handle self-refential formulae, including those with negative self-references. The idea is that the type $\triangleright A$ denotes the type of elements of A which are available after one time step. To it belong a constructor $\mathsf{next} : A \to \triangleright A$ and delayed application $- \circledast -$ as depicted in Fig. 1.3. We can then use the guarded fixpoint combinator $\mathsf{fix} : (\triangleright A \to A) \to A$ to get well-defined guarded recursive definitions.

Guarded fixpoints are unique and as such always both the largest and smallest fixpoint of an endofunctor. To exemplify how the guarded fixpoint combinator is used, we first introduce the type *guarded natural number streams* $\mathsf{gstr}_{\mathbb{N}}$. Without elaborating why it exists just yet, we note that it is the unique fixpoint of the functor

$$F_{\mathsf{gstr}_{\mathbb{N}}}(X) \triangleq \mathbb{N} \times \triangleright X \tag{1.2}$$

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{next}(a) : \triangleright A}$$

$$\frac{\Gamma \vdash f : \triangleright(A \to B) \qquad \Gamma \vdash a : \triangleright A}{\Gamma \vdash f \circledast a : \triangleright B}$$

$$\frac{\Gamma \vdash f : \triangleright A \to A}{\Gamma \vdash \mathsf{fix}(f) : A}$$

$$\begin{aligned} \mathsf{next}(f) \circledast \mathsf{next}(a) &= \mathsf{next}(f(a)) \\ (\mathsf{next}(\lambda x.x))(u) &= u \\ ((\mathsf{next}\circ) \circledast f \circledast g) \circledast a &= f \circledast (g \circledast a) \\ u \circledast \mathsf{next}(t) &= \mathsf{next}(\lambda g.g(t)) \circledast u \\ \mathsf{fix}(f) &= f(\mathsf{next}(\mathsf{fix}(f))) \end{aligned}$$

Figure 1.3: Interface of guarded type theory

Recalling the functor F_{stream_A} of Eq. (1.1) we observe that $F_{\mathsf{gstr}_{\mathbb{N}}} = F_{\mathsf{stream}_{\mathbb{N}}} \circ \triangleright$. Now this type has the constructor $\mathsf{cons} : (\mathbb{N} \times \triangleright \mathsf{gstr}_{\mathbb{N}}) \xrightarrow{\simeq} \mathsf{gstr}_{\mathbb{N}}$ which appends a natural number to a delayed guarded stream. We often write $a :: s$ instead of $\mathsf{cons}(a, s)$. There are furthermore functions $\mathsf{head} : \mathsf{gstr}_{\mathbb{N}} \to \mathbb{N}$ and $\mathsf{tail} : \mathsf{gstr}_{\mathbb{N}} \to \triangleright \mathsf{gstr}_{\mathbb{N}}$. We can now define a stream, which is constantly 0.

$$\mathsf{zeros} = \mathsf{fix}\,(\lambda\,(s : \triangleright \mathsf{gstr}_{\mathbb{N}}).0 :: s)$$

This is well-defined, because the term is well-typed, and not since the argument is nested under the $- :: -$ constructor. To illustrate this point, consider the function $\mathsf{merge}_A : \mathsf{gstr}_A \to \triangleright \mathsf{gstr}_A \to \mathsf{gstr}_A$ which merges two guarded A-streams by interleaving them. Note that the second stream can be delayed, as its first element is only needed for the second element of the resulting stream. We can now define a second stream of only 0, which composes the previously defined terms.

$$\mathsf{zeros}' \triangleq \mathsf{fix}\,(\lambda\,(s : \triangleright \mathsf{gstr}).\mathsf{merge}_{\mathbb{N}}(\mathsf{zeros})(s))$$

Again, this is well-defined, but since s is not nested under a constructor, a syntactic guardedness checker would sound the alarm.

The maps next and $- \circledast -$ turn $\triangleright$ into an applicative functor (see Fig. 1.3). They are very useful for applying the guarded hypothesis. For instance, to define a function $\mathsf{fmap} : (A \to B) \to \mathsf{gstr}_A \to \mathsf{gstr}_B$ by guarded recursion we proceed as follows

$$\text{let } G : \triangleright((A \to B) \to \mathsf{gstr}_A \to \mathsf{gstr}_B) \to (A \to B) \to \mathsf{gstr}_A \to \mathsf{gstr}_B \tag{1.3}$$

$$G\,F_{\mathsf{fmap}}\,f\,s_A \triangleq f(\mathsf{head}(s_A)) :: (F_{\mathsf{fmap}} \circledast (\mathsf{next}(f))) \circledast (\mathsf{tail}(s_A)) \text{ in} \tag{1.4}$$

$$\mathsf{fmap} \triangleq \mathsf{fix}(G : .\lambda) \tag{1.5}$$

Crucially, we apply the guarded hypothesis F_{fmap} to the delayed function $\mathsf{next}(f)$ and the tail of the guarded stream s_A under the later modality.

Semantics of Guarded Type theories In *guarded type theory* all types are intrinsically step-indexed, but instead of exposing the step indices directly to us, we interface with them through the later modality.

Birkedal et al. [23] showed that this *guarded type theory* extended with dependent types can be modeled in the topos of trees. The *topos of trees* is the category of presheaves over the first ordinal ω often denoted by $\mathsf{Pre}(\omega)$. Its objects $A : \mathsf{Pre}(\omega)$ are presheaves $A(1) \xleftarrow{r_1} A(2) \xleftarrow{r_2} \ldots$.

For example, there exists a presheaf of guarded natural number streams

$$\mathsf{gstr}_{\mathbb{N}} \triangleq \mathbb{N}^1 \leftarrow \mathbb{N}^2 \leftarrow \ldots \leftarrow \mathbb{N}^n \leftarrow \ldots$$

Observe that $\mathsf{gstr}_{\mathbb{N}}(n) = \mathbb{N}^n$, and thus a stream (a generalized element of the stream presheaf) $s : 1 \to \mathsf{gstr}_{\mathbb{N}}$ unveils its data step by step.

The authors of [23] observed that such recursive definitions can be written in the internal language of the topos. Indeed, there exists an endofunctor $\blacktriangleright$ together with a natural transformation $\mathsf{next} : \Gamma \to \blacktriangleright\, \Gamma$ (as depicted in Fig. 1.4), which allows us to present the presheaf $\mathsf{gstr}_{\mathbb{N}}$ as the unique solution to the recursive domain equation $X \simeq \mathbb{N} \times \blacktriangleright\, X$. In the topos of trees, solutions to *guarded recursive domain equations* exist for a large class of endofunctors, namely those that are *locally contractive* [23]. These definitions also descend into the slice categories and thus extend the internal dependent type theory of the topos by the later modality $\triangleright$.[2]

$$\begin{aligned} &\blacktriangleright : \mathsf{Pre}(\omega) \to \mathsf{Pre}(\omega) \\ &(\blacktriangleright A)(0) \triangleq 1 \\ &(\blacktriangleright A)(n+1) \triangleq A(n) \end{aligned}$$

$$\begin{array}{ccccc} A(0) & \xleftarrow[r_0]{} & A(1) & \xleftarrow[r_1]{} & \ldots \\ \downarrow{\scriptstyle !} & & \downarrow{\scriptstyle \mathsf{next}_0} & & \downarrow{\scriptstyle \mathsf{next}_1} \\ 1 & \xleftarrow[!]{} & A(0) & \xleftarrow[r_0]{} & \ldots \end{array}$$

Figure 1.4: The definition of the endofunctor $\blacktriangleright$

Universes Birkedal and Møgelberg [22] have shown that finding solutions to recursive domain equations can be internalized into the type theory. We start by extending a universe hierarchy $(\mathsf{U}_i, \mathsf{El}_i)$ by codes $\widehat{\triangleright}_i : \triangleright_{i+1} \mathsf{U}_i \to \mathsf{U}_i$ such that

$$\mathsf{El}_i(\widehat{\triangleright}_i A) = \triangleright_i(\mathsf{El}_i(A)).$$

However, in what follows we omit universe levels as well as applications of El and thus do not distinguish between *types* and *codes of types*.

In this setting, guarded recursive types can be constructed as fixpoints of functions. For example, $\mathsf{gstr}_{\mathbb{N}}$ can be defined as the following guarded fixpoint

$$\mathsf{gstr}_{\mathbb{N}} \triangleq \mathsf{fix}(\lambda\, (X : \triangleright\, \mathsf{U}).\mathbb{N} \times \triangleright X).$$

[2] For more information about the presheaf semantics of dependent type theory consult e.g. [62]

In the following, we use universes to define the categories of I-indexed types and functors on these categories. This allows us to classify a large class of endofunctors that have guarded fixpoints.

Definition 1.2.1. Given a type I, we let $I \to \mathsf{U}$ denote the *category* of I indexed types. Its objects are the objects of $I \to \mathsf{U}$ and the type of morphisms from X to Y is

$$X \to Y \triangleq \Pi(i : I).X(i) \to Y(i)$$

A *functor* $F : (I \to \mathsf{U}) \to (J \to \mathsf{U})$ is a pair of closed terms (F_0, F_1) which define actions on objects and morphisms, preserving identity and composition. (see [72, 81])

Now Birkedal and Møgelberg [22] showed the following theorem

Theorem 1.2.2. *Let F be an I-indexed endofunctor $(I \to \mathsf{U}) \to (I \to \mathsf{U})$, then there exists a guarded recursive type $\nu^g(F)$ satisfying*

$$\nu^g(F) \simeq F(\triangleright(\mathsf{next}(\nu^g(F)))),$$

where next *and $\triangleright$ are defined pointwise on families $X : I \to \mathsf{U}$. Furthermore, $\nu^g(F)$ is both an initial algebra as well as a final coalgebra for the endofunctor $F \circ \triangleright \circ \mathsf{next}$.*

For instance, the endofunctor F_{stream_A} of Section 1.1 can be defined as the 1-indexed endofunctor.

$$F_{\mathsf{stream}_A} \triangleq (F_0, F_1) : \mathsf{U} \to \mathsf{U} \qquad F_0(X) \triangleq A \times X \qquad F_1(f) \triangleq \mathsf{id}_A \times f$$

It follows from Theorem 1.2.2 that the guarded recursive type $\mathsf{gstr}_{\mathbb{N}}$ is precisely $\nu^g(F_{\mathsf{stream}_{\mathbb{N}}})$.

Limitations of guarded types While guarded recursion is very useful for constructing elements of guarded types, they can be a bit limited in usage. For instance, it is not possible to define a function $\mathsf{nth} : \mathsf{gstr}_{\mathbb{N}} \to \mathbb{N}$ which returns the n'th element of a guarded natural number stream. The only thing we can achieve is a function $\mathsf{nth} : \mathsf{gstr}_{\mathbb{N}} \to (\triangleright)^n\mathbb{N}$. This is because the later modality forces us to always preserve the number of computation steps. (Note that there cannot be a function $\mathit{wrong} : \triangleright\triangleright A \to \triangleright A$)

Therefore, *Clocked Type Theory* proposes a type former $\forall\kappa.A$ called *clock quantification*. Intuitively, it allows us to always go in a world where enough observations can be made. As a consequence, we get for instance that $\mathsf{nth} : \forall\kappa.\mathsf{gstr}_{\mathbb{N}} \to \mathbb{N}$ is definable.

1.3 Clocked Cubical Type Theory

The concrete type theory we use is Clocked Cubical Type Theory (CCTT) [72]. It combines Cubical Type Theory [37] with clocked type theory [80]. *Cubical Type Theory* realizes the ideas of homotopy type theory [106] and does so without disrupting canonicity — the property that ground types do not have exotic elements.

It is a dependent type theory with an additional primitive type — the *path type* $\mathsf{Path}_A\, a\, b$, which — following the homotopical intuition — serves as the identity type of CTT. In cubical type theory, all types carry the structure of n-dimensional cubes and this is reflected in the syntax, which includes additional values for each type namely the explicit *coercions* along paths and explicit *homogeneous compositions* which ensure that paths are symmetric and transitive.

Importantly, this structure ensures that isomorphic types are equal (their path type is inhabited) and therefore cubical type theory validates the univalence axiom [106]. This typal extensionality principle is particularly useful as it also implies function extensionality. Additionally, CTT also supports a novel scheme of inductive types: The higher inductive types.

Similarly to ordinary inductive types, higher inductive types can be declared by a list of constructors and come with an elimination principle that computes if applied to constructors. However, HITs may include path constructors, which in particular allows us to write down definitions of propositional truncation as well as effective quotient types.

Clocked type theory on the other hand is a guarded type theory where every step-indexed type is parametrized by a *clock*. Extending the context by a clock is a novel context formation rule, as is the addition of *ticks* to a specific clock. Thereby, types can be simultaneously step-indexed over multiple clocks. More importantly for us, it adds the type former $\forall \kappa.A$ (clock quantification) which can be used to realize coinductive types.

The combination of these two type theories furthermore validates the very useful extensionality principle for guarded types [24], which tells us that a delayed path between two points corresponds to a path between the delayed points.

$$\triangleright^\kappa(\mathsf{Path}_A\ a\ b) \simeq \mathsf{Path}_{\triangleright^\kappa A}\ \mathsf{next}(a)\ \mathsf{next}(b)$$

For this work, we do not benefit from the intrinsic structure of CCTT-types per se and are only interested in derived properties such as the extensionality principles for function types and the $\triangleright^\kappa$ type. We furthermore use higher inductive types to define the finite distribution monad, which is needed to interpret probabilistic choice, as well as the propositional truncation, which allows us to reason proof irrelevantly in CCTT.

CCTT is implemented in Agda and has already been used in [88].

In what follows, we will give an incomplete presentation of CCTT, only showcasing the features we make use of.

Cubical Type Theory

As discussed previously, the idea of cubical type theory is to equip types with a structure that ensures that every construction respects *paths*. Consequently, we consider terms connected by a path to be equals. In classical homotopy theory, a path is simply a continuous map from the interval into a topological space.

Therefore, CTT (and by inheritance CCTT) introduces the cubical interval $\mathbb{I}$. While $\mathbb{I}$ is not a type, there is a judgment $\Gamma \vdash r : \mathbb{I}$ saying that r is of the form

$$r,s ::= i0 \mid i1 \mid i \mid 1-r \mid r \wedge s \mid r \vee s.$$

Intuitively, one can think of $\mathbb{I}$ being the real interval with endpoints 0 and 1 as well as the minimum $r \wedge s$ and maximum $r \vee s$.[3]

$\Gamma \vdash \lambda(i : \mathbb{I}).t : \mathsf{Path}_A\ a\ b$ denotes a function from $\mathbb{I}$ to A where the endpoints are fixed by $t[i0/i] = a$ and $t[i1/i] = b$. In other words, it denotes a *path* between a and b.

As discussed before, we consider $\mathsf{Path}_A\ a\ b$ to be the correct notion of intensional equality in this system and therefore we write $a =_A b$ instead of $\mathsf{Path}_A\ a\ b$. If no confusion arises, we omit the subscript of the equality. Formally, paths validate beta and eta rules just as functions, but additionally a path $p : a =_A b$ reduces to the boundary declared in its type for $i0$ and $i1$.

$$(\lambda i.t)r \equiv t[r/i] \qquad p \equiv \lambda i.p(i) \qquad p(i0) \equiv a \qquad p(i1) \equiv b$$

The proof of reflexivity is simply the constant function $\lambda(i : \mathbb{I}).a : a = a$. Furthermore, it is easy to prove function extensionality, since if $p : \Pi(a : A).(f(a) = g(a))$, then $\lambda i.\lambda a.p(a)(i) : f =_{\Pi(a:A).B} g$.

Following the Leibniz principle, equal terms should be indiscernible. Hence, we expect that if $p : a = b$ and we have a proof of $P(a)$ for some predicate P, then also $P(b)$ should be provable. Note, that such a principle (often referred to as *transport*) is in the absence of the standard elimination rule of intensional identity types not admissible (just as the transitivity of equality).

It turns out that both transport and transitivity are special cases of the *composition* operation. To formulate it, we need to discuss partial terms first. Cubical type theories have a special primitive type of propositions, the *face lattice* $\mathbb{F}$. Its elements, the *face formulas*, are generated by the grammar

$$\varphi, \psi ::= 0_{\mathbb{F}} \mid 1_{\mathbb{F}} \mid (i = i0) \mid (i = i1) \mid \varphi \wedge \psi \mid \varphi \vee \psi$$

where $(i = i0) \wedge (i = i1) \equiv 0_{\mathbb{F}}$. We call a term partial if it is defined in a context extended by a face formula. The notation $\Gamma \vdash u : A[\varphi \mapsto v]$ means that $\Gamma \vdash u : A$ extends the partial term v as expressed by $\Gamma, \varphi \vdash u \equiv v : A$. Now the composition operation states that this extensibility is preserved along paths:

$$\frac{\Gamma \vdash \varphi : \mathbb{F} \qquad \Gamma, i : \mathbb{I} \vdash A \qquad \Gamma, \varphi, i : \mathbb{I} \vdash u : A \qquad \Gamma \vdash a_0 : A[i0/i][\varphi \mapsto u[i0/i]]}{\Gamma \vdash \mathsf{comp}^i A\ [\varphi \mapsto u]\, a_0 : A[i1/i][\varphi \mapsto u[i1/i]]}$$

We can now for instance define transport as

$$\frac{\Gamma, i : \mathbb{I} \vdash A \qquad \Gamma \vdash a : A[i0/i]}{\Gamma \vdash \mathsf{transp}^i A\, a \triangleq \mathsf{comp}^i A\, [0_{\mathbb{F}} \mapsto -]\, a : A[i1/i]}$$

For further reading on the face lattice and compositions see [37].

[3] Actually, we have to require additional coherences to turn $\mathbb{I}$ into a de Morgan algebra.

Higher Inductive Types Higher inductive types (HITs) generalize ordinary inductive types by allowing us to specify *higher-dimensional constructors*. Phrased differently, this allows us to inductively define a type together with coherences that we want to hold. This is very useful, for instance, quotient types become the special case of 0-*truncated* (or *set-truncated*) HITs. We use the scheme of Cavallo and Harper [32] which was generalized by Kristensen et al. [72] to integrate into clocked cubical type theory. We can declare constructors of higher inductive types by a tuple

$$(constr, (args; rec\_args; \mathbb{I}args; \mathbb{F}args)) \text{ where}$$

- $constr$ is the name of the constructor
- $args$ are the non-recursive arguments and rec_args are the recursive arguments of the constructor
- $\mathbb{I}args$ are the interval arguments. To a first approximation, the more interval variables are included, the higher the dimension of the constructor gets.
- $\mathbb{F}args$ denotes the boundary conditions for paths of the form $\begin{bmatrix} i = i0 \mapsto t_0 \\ i = i1 \mapsto t_1 \end{bmatrix}$. Importantly, the terms t_0 and t_1 may depend on all recursive and non-recursive arguments as well as the constructors declared before. Note that in general, these conditions can be more complicated, but for us this is sufficient.

For a much more in-depth explanation consult Cavallo and Harper [32].

Propositions and Sets Our first example of a HIT is the propositional truncation, which squashes an arbitrary type to a *homotopy propositions* (hPropositions). A homotopy proposition is a type with at most one element. This captures the idea of proof irrelevance — either a proposition is provable or not, it does not matter how we prove it.

Definition 1.3.1. Given a type A we write $||A||_{-1}$ to denote the type defined by the constructors

- $(|-|, (A; \cdot; \cdot; []))$
- $\left(\mathsf{isprop}, \left(\cdot; (a_0, a_1 : ||A||_{-1}); (i : \mathbb{I}); \begin{bmatrix} i = i0 \mapsto a_0 \\ i = i1 \mapsto a_1 \end{bmatrix}\right)\right)$

In what follows we will use a more intuitive notation and define HITs as lists of typed constructors. For the propositional truncation, we get

$$\begin{aligned} &(A : \mathsf{U}) \vdash \textbf{ inductive } ||A||_{-1} : \mathsf{U} \textbf{ where} \\ &|-| : A \to ||A||_{-1} \\ &\mathsf{isprop} : (a_0, a_1 : ||A||_{-1}) \to a_0 = a_1 \end{aligned}$$

Note that the path type $a_0 =_{||A||_{-1}} a_1$ internalizes functions from $\mathbb{I}$ to $||A||_{-1}$ with boundaries $i = i0 \mapsto a_0$ and $i = i1 \mapsto a_1$. Thus, one can easily recover the correct constructor declaration from the types.

The elimination principle of $||A||_{-1}$ lifts a function $A \to B$ to a function $||A||_{-1} \to B$ as long as B is also a proposition.

The propositional truncation allows us to express ordinary logic with operators in type theory. The empty type 0 and the one element type 1 are already homotopy propositions and the non-dependent product $\varphi \times \psi$ is a proposition if φ and ψ are. We define $\forall(x:A).\varphi \triangleq \Pi(x:A).\varphi(x)$ which is a proposition if φ is. Coproduct types and dependent sums do not preserve propositions and thus we let $\varphi \vee \psi \triangleq ||\varphi + \psi||_{-1}$ and $\exists(x:A).\varphi(x) \triangleq ||\Sigma_{(x:A)}\varphi||_{-1}$.

We denote the universe of propositions by

$$\begin{aligned} \mathsf{Prop}_i &\triangleq (\varphi : \mathsf{U}_i) \times \mathsf{isProp}\ \varphi \ \textbf{where} \\ \mathsf{isProp}\ \varphi &\triangleq (x,y:\varphi) \to (x = y). \end{aligned}$$

For a thorough introduction to logic in type theory consult Univalent Foundations Program [106].

Homotopy sets (or short hSets) are types for which all path types (and thus all higher types) are trivial. There is a similar truncation $||-||_0$ as for propositions, in fact there is a corresponding truncation for every *h-level*.

Finite Distribution Monad For our second paper, we interpret a probabilistic choice operator using a special type of monad, the *convex delay algebras*. The construction uses a monad $\mathscr{D}$ for finite distributions. In classical probability theory, the finite distributions over a set A are defined as maps with finite support $\mu : A \to [0,1]$ whose values sum up to 1. This definition can be internalized in type theory as a sum type, but it is unclear how to define a monad structure without assuming that A has decidable equality.

Therefore, we represent $\mathscr{D}$ as the free monad for the theory of *convex algebras* [66], which is an equivalent definition.

To do so, we first have to discuss the open rational unit interval $(0,1)$. There are different approaches on how to internalize such a type. For instance, we could represent $(0,1)$ as coprime natural number pairs (n,d) such that $n < d$. We leave the details of the implementation implicit but note that however $(0,1)$ is implemented, we require the following:

Proposition 1.3.2. *$(0,1)$ is closed under the operations:*

- *for all $p,q : (0,1)$ we have that $pq : (0,1)$.*
- *for all $p : (0,1)$ it follows that $1-p : (0,1)$.*
- *$(0,1)$ is closed under convex combinations: For all $p,q_1,q_2 : (0,1)$ we have that $pq_1 + (1-p)q_2 : (0,1)$*

- *For all* $p, q : (0,1)$ *we have that* $\frac{p}{p+q} : (0,1)$.

Multiplication and addition commute, associate and distribute as usual.

Definition 1.3.3 (Convex Algebra)**.** A *convex algebra* is a set A together with an operation $- \oplus - : (0,1) \to A \to A \to A$ such that

$$\mu \oplus_p \mu = \mu \tag{idem}$$
$$\mu \oplus_p \nu = \nu \oplus_{1-p} \mu \tag{comm}$$
$$(\mu_1 \oplus_p \mu_2) \oplus_q \mu_3 = \mu_1 \oplus_{pq} (\mu_2 \oplus_{\frac{q-pq}{1-pq}} \mu_3) \tag{assoc}$$

A *convex algebra homomorphism* is a map $f : A \to B$ between convex algebras such that $f(\mu \oplus_p \nu) = f(\mu) \oplus_p f(\nu)$.

Definition 1.3.4. The free convex A-algebra $\mathscr{D}(A)$ can be represented as a HIT by the following signature (informal presentation):

$$\begin{aligned}
\delta &: A \to \mathscr{D}(A) \\
- \oplus - &: (0,1) \to \mathscr{D}(A) \to \mathscr{D}(A) \to \mathscr{D}(A) \\
\mathsf{idem} &: \forall p : (0,1).\ \forall \mu : \mathscr{D}(A).\ (\mu \oplus_p \mu = \mu) \\
\mathsf{comm} &: \forall p : (0,1).\ \forall \mu, \nu : \mathscr{D}(A).\ (\mu \oplus_p \nu = \nu \oplus_{1-p} \mu) \\
\mathsf{assoc} &: \forall p, q.\ \forall \mu_1, \mu_2, \mu_3.\ \left((\mu_1 \oplus_p \mu_2) \oplus_q \mu_3 = \mu_1 \oplus_{pq} (\mu_2 \oplus_{\frac{q-pq}{1-pq}} \mu_3)\right) \\
\mathsf{isset} &: \forall \mu, \nu : \mathscr{D}(A).\ \forall eq_1, eq_2 : \mu = \nu.\ (eq_1 = eq_2)
\end{aligned}$$

The final constructor set-truncates the type.

The recursion principle for HITs corresponds precisely to $\mathscr{D}(A)$ being the free algebra which is why the following proposition is immediate.

Proposition 1.3.5. *For every set* A *and convex algebra* B *and map* $f : A \to B$ *we get a unique convex algebra homomorphism* $\overline{f} : \mathscr{D}(A) \to B$ *such that* $f = \overline{f} \circ \delta$. *Consequently,* $\mathscr{D}$ *forms a monad on the category of sets.*

It is easy to see how elements of convex algebras represent finite distributions. Intuitively speaking, $\delta(a)$ denotes the Dirac distribution which has all weight on a. The convex combination $d \oplus_q d'$ is equal to d with probability q and equal to d' with probability $1 - q$.

Considering this, it is straightforward to define a function $\mathsf{Prob}_\mathsf{A} : \mathscr{D}(A) \to A \to [0,1]$, which computes the probability that a distribution assumes a certain value.

Clocked Type Theory

Clocked Cubical Type theory extends CTT with multiclocked guarded recursion [80]. The $\triangleright$ modality is indexed by a clock κ. Such clocks are either variables of the pretype clock or the clock constant κ_0. Similar to the cubical interval, clock is not a type but

Contexts :

$$\frac{\Gamma \vdash \qquad \kappa : \mathsf{clock} \notin \Gamma}{\Gamma, \kappa : \mathsf{clock} \vdash} \qquad \frac{\kappa : \mathsf{clock} \in \Gamma \qquad \alpha \notin \Gamma}{\Gamma, \alpha : \kappa \vdash} \qquad \frac{\alpha \notin \Gamma}{\Gamma, \alpha : \kappa_0 \vdash}$$

Types :

$$\frac{\Gamma, \alpha : \kappa \vdash A}{\Gamma \vdash \triangleright(\alpha : \kappa).A} \qquad \frac{\Gamma, \kappa : \mathsf{clock} \vdash A}{\Gamma \vdash \forall \kappa.A}$$

Terms :

$$\frac{\Gamma, \alpha : \kappa \vdash t : A}{\Gamma \vdash \lambda(\alpha : \kappa).t : \triangleright(\alpha : \kappa).A} \qquad \frac{\Gamma \vdash t : \triangleright(\alpha : \kappa).A \qquad \Gamma, \beta : \kappa, \Gamma' \vdash}{\Gamma, \beta : \kappa, \Gamma' \vdash t\,[\beta] : A\,[\beta/\alpha]}$$

$$\frac{\Gamma \vdash t : \triangleright^{\kappa} A \to A}{\Gamma \vdash \mathsf{dfix}^{\kappa} t : \triangleright^{\kappa} A} \qquad \frac{\Gamma \vdash t : \triangleright^{\kappa} A \to A}{\Gamma \vdash \mathsf{pfix}^{\kappa} t : \triangleright(\alpha : \kappa).\,(\mathsf{dfix}^{\kappa} t[\alpha] = t(\mathsf{dfix}^{\kappa} t))}$$

$$\frac{\Gamma, \kappa : \mathsf{clock} \vdash t : A}{\Gamma \vdash \Lambda\kappa.t : \forall \kappa.A} \qquad \frac{\Gamma \vdash t : \forall \kappa.A \qquad \Gamma \vdash \kappa' : \mathsf{clock}}{\Gamma \vdash t[\kappa'] : A[\kappa'/\kappa]}$$

Figure 1.5: Selection of typing rules for CCTT

$$\lambda\kappa.t\,[\kappa] \equiv t \qquad \lambda(\alpha : \kappa).(t\,[\alpha]) \equiv t$$
$$(\lambda\kappa.t)\,[\kappa'] \equiv t[\kappa'/\kappa] \qquad (\lambda(\alpha : \kappa).t)\,[\alpha'] \equiv t[\alpha'/\alpha]$$

Figure 1.6: Beta and Eta rules for modal types

only occurs in a novel context extension rule (Fig. 1.5). Clocks can be quantified using the type former $\forall\kappa.A$, which implements ideas proposed by Atkey and McBride [13] to realize coinductive types. The rules are similar to those of Π-types and $\forall\kappa.A$ also validates a functional extensionality principle. Furthermore, it is an applicative functor as summarized in Fig. 1.7.

$$\frac{f : \forall\kappa.A \to B \qquad a : \forall\kappa.A}{\Lambda\kappa.f[\kappa](a[\kappa]) : \forall\kappa.B} \qquad \frac{\forall\kappa.a =_A b}{\Lambda\kappa.a =_{\forall\kappa.A} \Lambda\kappa.b}$$

Figure 1.7: Admissible rules for $\forall\kappa$

The introduction and elimination of $\triangleright(\alpha : \kappa).$ are by tick abstraction and application, similar to the tick calculus of Bahr et al. [14]. Extending a context by a tick $\Gamma, \alpha : \kappa$ is a context extension rule, that hypothesizes that time has passed on clock κ. In the context $\Gamma, \alpha : \kappa, \Gamma'$, the tick α signals that time has passed between the values represented by the variables in Γ and the values represented by variables in Γ'.

It is relatively convenient and intuitive to program using ticks. Consider for example the following definition:

$$f : \Sigma_{(x:\triangleright^\kappa A)} \left(\triangleright(\alpha : \kappa).B(x[\alpha])\right) \to \triangleright^\kappa(\Sigma_{x:A} B(x))$$
$$f(x,y) \triangleq \lambda(\alpha:\kappa).(x[\alpha], y[\alpha]).$$

We first abstract the tick $\alpha : \kappa$, and then need to define a term of type $\Sigma_{x:A} B(x)$. Since we have $x : \triangleright^\kappa A$ and $y : \triangleright(\alpha : \kappa).B(x[\alpha])$ we can simply apply the tick α to each of them and get the result. Note that the tick application rule in Fig. 1.5 requires that β (and any other variable listed afterward in the context) does not occur freely in $t : \triangleright(\alpha : \kappa).A$. This ensures that the number of steps is preserved and we cannot apply the same tick twice. The following would thus be ill-typed:

$$\mathsf{wrong} : \triangleright^\kappa \triangleright^\kappa A \to \triangleright^\kappa A$$
$$\mathsf{wrong}\, a \triangleq \lambda(\alpha:\kappa).a\,[\alpha]\,[\alpha].$$

Along with several rules of standard Martin-Löf type theory, we also omitted many technical rules governing clocks and ticks. For instance, CCTT allows us to hypothesize certain timeless assumptions that are needed to prove the extensionality principle for $\triangleright^\kappa$

$$x =_{\triangleright^\kappa A} y \simeq \triangleright(\alpha : \kappa).x[\alpha] =_A y[\alpha] \tag{1.6}$$

This principle implies that $\triangleright^\kappa$ preserves h-levels and in particular sets and propositions.

Further rules left out are the *simple* and *forcing* tick substitutions. These let us define the *force* operator of Atkey and McBride [13].

$$\mathsf{force} : \forall\kappa.\triangleright^\kappa A \to \forall\kappa.A \tag{1.7}$$

Finally, in CCTT we can inhabit the *tick irrelevance axiom*

$$\mathsf{tirr}^\kappa : (x : \triangleright^\kappa A) \to \triangleright(\alpha : \kappa).\triangleright(\beta : \kappa).x[\alpha] =_A x[\beta],$$

which is needed to prove that Eq. (1.7) is an equivalence of types.

Guarded recursion

The tick rules can be used to implement the next and $- \circledast^\kappa -$ operator from before, to turn each $\triangleright(\alpha : \kappa).$ into the applicative "later" modality of Fig. 1.3.

$$\mathsf{next}^\kappa(a) \triangleq \lambda(\alpha:\kappa).a \qquad\qquad f \circledast^\kappa a \triangleq \lambda(\alpha:\kappa).f\,[\alpha](a\,[\alpha])$$

In practice, it is more convenient to use ticks directly instead of interfacing with next and $- \circledast -$. As the archetypical ingredient of guarded type theories, we have a delayed guarded fixpoint constructor dfix^κ and its propositional unfolding rule pfix^κ (see Fig. 1.5). We recover the guarded fixpoint combinator of Fig. 1.3 by defining

$$\frac{\Gamma \vdash t : \triangleright^\kappa A \to A}{\Gamma \vdash \mathsf{fix}^\kappa t \triangleq t(\mathsf{dfix}^\kappa t) : A}$$

Using the propositional unfolding rule pfix^κ (see Fig. 1.5), one can prove that

$$\mathsf{fix}^\kappa t = t(\mathsf{next}^\kappa(\mathsf{fix}^\kappa t)). \tag{1.8}$$

Remark 3. Most of the time, we define guarded recursive terms through equations, which uniquely determine them. For instance, we can redefine the function in Eq. (1.5) by writing

$$\mathsf{fmap}^\kappa f\, s = f(\mathsf{head}^\kappa(s)) :: \lambda(\alpha : \kappa).\mathsf{fmap}^\kappa f\,(\mathsf{tail}^\kappa(s)\,[\alpha])$$

While this is not the correct application of the guarded fixpoint combinator, once fmap^κ is defined, it will satisfy this equality by Eq. (1.8). For this to work, the recursive call of the function needs to occur under a tick.

Similar to Theorem 1.2.2, for any I-indexed endofunctor F and clock κ we get guarded fixpoints $\nu^\kappa(F)$ such that

$$\nu^\kappa(F) \simeq F(\triangleright^\kappa(\mathsf{next}^\kappa(\nu^\kappa(F)))).$$

Guarded Partiality Monad We revisit the partiality monad of Example 2, which ought to be the final coalgebra of the endofunctor $F_\mathsf{L} : (\mathsf{U} \to \mathsf{U}) \to \mathsf{U} \to \mathsf{U}$

$$F_\mathsf{L} \triangleq \lambda G.\lambda A.(A + G(A)).$$

Algebras of the functor $F_\mathsf{L} \circ \triangleright^\kappa \circ \mathsf{next}^\kappa$ are called *guarded delay algebras* and can be characterized as follows:

Definition 1.3.6 (Guarded delay algebras)**.** A (κ-)delay algebra is a set A together with a map $\mathsf{step}^\kappa : \triangleright^\kappa A \to A$. A *delay algebra homomorphism* is a map $f : A \to B$ such that $f(\mathsf{step}^\kappa(a)) = \mathsf{step}^\kappa(\lambda(\alpha : \kappa).f(a\,[\alpha]))$.

Example 4. The universe U_i together with $\widehat{\triangleright_i^\kappa}$ forms a guarded delay algebra. Similarly, the universe of propositions Prop_i and $\widehat{\triangleright_i^\kappa}$ form a delay algebra, as $\triangleright^\kappa$ preserves propositions.

Let $\mathsf{L}^\kappa \triangleq \nu^\kappa(F_\mathsf{L})$ denote the unique guarded fixpoint of F_L. Now Theorem 1.2.2 implies that for any type $A : \mathsf{U}$ we have

$$\mathsf{L}^\kappa(A) \simeq A + \mathsf{L}^\kappa(A) \tag{1.9}$$

The map $\mathsf{step}^\kappa(a) \triangleq \mathsf{inr}\,(a)$ witnesses that $\mathsf{L}^\kappa(A)$ is indeed a delay algebra and $\eta^\kappa(a) \triangleq \mathsf{inl}\,(a)$ embeds A into $\mathsf{L}^\kappa(A)$.

Example 5 (Guarded Loop). For any type A, we define the infinite, unproductive loop $\bot^\kappa : \mathsf{L}^\kappa(A)$ by letting $\bot^\kappa \triangleq \mathsf{fix}^\kappa \mathsf{step}^\kappa$.

Lemma 1.3.7 (Freeness of L^κ)**.** *For any set $A : \mathsf{U}$ the type $\mathsf{L}^\kappa(A)$ is the free guarded delay A-algebra. Thus, for any delay algebra B and map $f : A \to B$ there is a unique lift $\overline{f} : \mathsf{L}^\kappa(A) \to B$.*

Proof. This proof is a consequence of Theorem 1.2.2, but it is instructive to prove it by hand.

Suppose $f : A \to B$ and (B, step_B) is a delay algebra. We define the lift $\overline{f} : \mathsf{L}^\kappa(A) \to B$ by guarded recursion, and thus it suffices to define a function $\triangleright^\kappa(\mathsf{L}^\kappa(A) \to B) \to \mathsf{L}^\kappa(A) \to B$.

Hence, let $g : \triangleright^\kappa(\mathsf{L}^\kappa(A) \to B)$ and considering Eq. (1.9) we continue by casing on $A + \triangleright^\kappa \mathsf{L}^\kappa(A)$ to define a function $\mathsf{L}^\kappa(A) \to B$.

$$\begin{aligned} \overline{f}(\eta^\kappa(a)) &= f(a) \\ \overline{f}(\mathsf{step}^\kappa(a)) &= \mathsf{step}_B(\lambda(\alpha : \kappa).g[\alpha](a[\alpha])) \end{aligned}$$

For *uniqueness* we show that any other f-lift h is pointwise equal to $\overline{f}$. Then function extensionality implies the result. We again proceed by guarded recursion and thus assume $p : \triangleright^\kappa(\Pi(x : \mathsf{L}^\kappa(A))(h(a) = \overline{f}(a)))$. The $\eta^\kappa a$-case is immediate since both h and $\overline{f}$ reduce to f. For any $a : \triangleright^\kappa \mathsf{L}^\kappa(A)$ we get that

$$\lambda(\alpha : \kappa).p[\alpha](a[\alpha]) : \triangleright^\kappa(h(a) = \overline{f}(a)),$$

which by Eq. (1.6) is equivalent to

$$\lambda(\alpha : \kappa).h(a[\alpha]) = \lambda(\alpha : \kappa).\overline{f}(a[\alpha]).$$

Therefore, the $\mathsf{step}^\kappa(a)$-case follows since

$$\begin{aligned} h(\mathsf{step}^\kappa a) &= \mathsf{step}_B(\lambda(\alpha : \kappa).h(a[\alpha])) \\ &= \mathsf{step}_B(\lambda(\alpha : \kappa).\overline{f}(a[\alpha])) \\ &= \overline{f}(\mathsf{step}^\kappa a) \end{aligned}$$

□

We often write $t >\!\!>\!\!=^\kappa f$ for $\overline{f}(a)$ where $\overline{f}$ is the unique extension of f to a delay algebra homomorphism.

Corollary 1.3.8 (Lifting of Predicates)**.** *Since* Prop *and* $\triangleright^\kappa$ *form a delay algebra we can unqiuely extend predicates $Q : A \to \mathsf{Prop}$ to $\mathsf{L}^\kappa(Q) : \mathsf{L}^\kappa(A) \to \mathsf{Prop}$.*

Next, we note that L^κ is a functor and lifts morphism $f : A \to B$ to $\mathsf{L}^\kappa(f) : \mathsf{L}^\kappa(A) \to \mathsf{L}^\kappa(B)$ by letting

$$\mathsf{L}^\kappa(f) \triangleq \overline{\eta^\kappa \circ f}$$

Lemma 1.3.9 (Functoriality of L^κ)**.** *Let $f : A \to B$ and $g : B \to C$, then*

$$\mathsf{L}^\kappa(g \circ f) = \mathsf{L}^\kappa(g) \circ \mathsf{L}^\kappa(f)$$

Proof. Instead of reiterating the proof by guarded recursion, we simply note that either function is the unique extension of $\eta^\kappa \circ g \circ f : A \to \mathsf{L}^\kappa(C)$. □

Lastly, L^κ forms a monad with the monad multiplication $\mu_A : \mathsf{L}^\kappa(\mathsf{L}^\kappa(A)) \to \mathsf{L}^\kappa(A)$ being the unique delay algebra homomorphism that extends the identity function $\mathsf{id} : \mathsf{L}^\kappa(A) \to \mathsf{L}^\kappa(A)$.

Lemma 1.3.10 (Monad laws)**.** *L^κ with unit η^κ and multiplication μ^κ forms a monad.*

Proof. We have to show that for any $X : \mathsf{U}$ the maps $\mu_X \circ \mathsf{L}^\kappa(\mu_X)$ and $\mu_X \circ \mu_{\mathsf{L}^\kappa(X)}$ are equal. This is however immediate, since both functions extend $\mu_X : \mathsf{L}^\kappa(\mathsf{L}^\kappa(X)) \to \mathsf{L}^\kappa(X)$. Indeed, a simple calculation reveals that

$$\mu_X \circ (\mathsf{L}^\kappa(\mu_X))(\eta^\kappa(x)) = \mu_X(\eta^\kappa(x)) = \mu_X \circ \mu_{\mathsf{L}^\kappa(X)}(\eta^\kappa(x))$$

Similarly, it follows that $\mu_X \circ \eta^\kappa_{\mathsf{L}^\kappa(X)}$ is equal to $\mu_X \circ \mathsf{L}^\kappa(\eta^\kappa_X)$ since either function is equal to $\overline{\eta^\kappa_X}$. Note that $\overline{\eta^\kappa_X}$ is the identity function. □

Guarded Convex Delay Algebra The binary choice operator $\mathsf{choice}^p(M,N)$ evaluates with probability p to M and with probability $1-p$ to N. Consequently, programs of $\mathsf{FPC}_\oplus$ evaluate to distributions over values, not just values. Combined with general recursion, we can program procedures, which furthermore evaluate to distributions with infinite support, such as the geometric distribution. Consequently, to interpret $\mathsf{FPC}_\oplus$ we need more than the partiality monad.

Definition 1.3.11 (Convex Delay Algebra)**.** A set A is a *convex delay algebra* if it is both a *delay algebra* and a *convex algebra*.

We can define the *guarded convex delay algebra* as the guarded recursive type satisfying

$$\mathsf{D}^\kappa A \simeq \mathscr{D}(A + \triangleright^\kappa(\mathsf{D}^\kappa A)) \tag{1.10}$$

Proposition 1.3.12. *For any set A, $\mathsf{D}^\kappa A$ denotes the free convex delay A-algebra.*

Using guarded recursion, we can define a process generating a geometric distribution starting at value n.

$$\mathsf{geo}^\kappa_p : \mathbb{N} \to \mathsf{D}^\kappa(\mathbb{N}) \tag{1.11}$$

$$\mathsf{geo}^\kappa_p(n) = (\delta^\kappa n) \oplus_p \mathsf{step}^\kappa(\lambda(\alpha : \kappa).\mathsf{geo}^\kappa_p(n+1)) \tag{1.12}$$

This is well-defined since the recursive call occurs under a tick (recall Remark 3). By construction, it unfolds recursively to

$$\mathsf{geo}^\kappa_p(0) = (\delta^\kappa 0) \oplus_p \mathsf{step}^\kappa(\lambda(\alpha : \kappa).\mathsf{geo}^\kappa_p(1))$$

$$= (\delta^\kappa 0) \oplus_p \left(\mathsf{step}^\kappa \lambda(\alpha : \kappa).\left((\delta^\kappa 1) \oplus_p \mathsf{step}^\kappa \lambda(\beta : \kappa).\mathsf{geo}_p^\kappa(2)\right)\right)$$
$$= \ldots$$

The guard $\triangleright^\kappa$ prevents us from computing the probability of termination. This is why the correct monad to investigate probabilistic processes is a *coinductive convex delay monad* $\mathsf{D}^\forall A$.

Coinductive Types

As discussed in Section 1.1 we set out to define coinductive types, which represent the greatest fixpoints of endofunctors. This is useful to model infinitary processes.

CCTT realizes coinductive types for endofunctors that commute with clock quantification.

Definition 1.3.13. A functor F commutes with clock quantification if for all families $X : I \to \mathsf{U}$ the canonical map

$$F(\forall \kappa.X) \to \forall \kappa.F(X)$$

is an equivalence

To exemplify this technique, we return to the example of streams. Recall that $\mathsf{gstr}_{\mathbb{N}}^\kappa$ is the guarded fixpoint of $F_{\mathsf{gstream}_{\mathbb{N}}}$ (see Eq. (1.2)), satisfying the guarded domain equation

$$\mathsf{gstr}_{\mathbb{N}}^\kappa \simeq \mathbb{N} \times \triangleright^\kappa \mathsf{gstr}_{\mathbb{N}}^\kappa.$$

As discussed in Eq. (1.1), we are however looking for $\nu(F_{\mathsf{stream}_{\mathbb{N}}})$. Let us assume for the moment that $F_{\mathsf{gstream}_{\mathbb{N}}}$ commutes with clock quantification. Consequently, we get that

$$\begin{aligned}\forall \kappa.\mathsf{gstr}_{\mathbb{N}}^\kappa &\simeq \forall \kappa.(\mathbb{N} \times \triangleright^\kappa \mathsf{gstr}_{\mathbb{N}}^\kappa)\\ &\simeq \mathbb{N} \times \forall \kappa.(\triangleright^\kappa \mathsf{gstr}_{\mathbb{N}}^\kappa)\\ &\simeq \mathbb{N} \times \forall \kappa.\mathsf{gstr}_{\mathbb{N}}^\kappa,\end{aligned}$$

where the last step follows from the isomorphism induced by Eq. (1.7). Thus, $\forall \kappa.\mathsf{gstr}_{\mathbb{N}}^\kappa$ is a coalgebra of $F_{\mathsf{stream}_{\mathbb{N}}}$. The fact that it is the final one is a corollary of a more general property:

Theorem 1.3.14. *Let F be an endofunctor which commutes with clock quantification, and let $\nu^\kappa(F)$ be the guarded recursive type satisfying $\nu^\kappa(F) \simeq F(\triangleright^\kappa(\nu^\kappa(F)))$. Then $\nu(F) \triangleq \forall \kappa.\nu^\kappa(F)$ has a final F-coalgebra structure.*

A proof of this theorem can be found in Møgelberg [81].

Corollary 1.3.15. $\forall \kappa.\mathsf{gstr}_{\mathbb{N}}^\kappa$ *is the final coalgebra* $\nu(F_{\mathsf{stream}_{\mathbb{N}}})$.

Proof. We have that $F_{\mathsf{gstream}_{\mathbb{N}}} = F_{\mathsf{stream}_{\mathbb{N}}} \circ \triangleright^\kappa$ and thus the theorem applies. □

For this to be useful, we need a large collection of functors which commute with clock quantification. To that goal, we first introduce the concept of *clock irrelevant* types.

Definition 1.3.16. A type A is *clock-irrelevant* if the canonical map $A \to \forall\kappa.A$ (for κ fresh) is an isomorphism.

There is a large class of such *clock-irrelevant* types which are described in detail in [72]. It includes in particular the natural numbers $\mathbb{N}$ and is closed under various type formers, such as dependent products and sums, coproducts, clock quantification and path types. Furthermore, all propositions are clock-irrelevant, since by applying the clock constant κ_0 we get a map $\forall\kappa.P \to P$ for all propositions.

The following theorem of Kristensen et al. [72] identifies a large number of functors which commute with clock quantification.

Theorem 1.3.17. *The collection of endofunctors commuting with clock quantification is closed under composition, pointwise product, pointwise* Π*, pointwise* Σ*, over clock irrelevant types, and pointwise universal quantification over clocks. If* F *commutes with clock quantification then the guarded recursive type* X *satisfying* $X \simeq F(\triangleright^\kappa X)$ *is clock irrelevant. Any path type of clock irrelevant type is clock irrelevant.*

Furthermore, using the notion of *induction under clocks* Kristensen et al. [72] established that many functors involving HITs also commute with clock quantification. In particular, all inductive types are clock irrelevant, but specific higher inductive types such as the finite powerset monad commute with clock quantification as well ([72]). As we will see in §3, this also holds for the finite distribution monad.

Proposition 1.3.18. *The finite distribution monad* $\mathscr{D}$ *commutes with clock quantification.*

Caprettas Partiality Monad Revisited We define $\mathsf{L}(A) \triangleq \forall\kappa.\mathsf{L}^\kappa(A)$. If A is clock irrelevant, then Theorem 1.3.14 implies that $\mathsf{L}(A)$ is the final coalgebra $\nu(F_\mathsf{L})$ of F_L, where F_L is the functor of Example 2.

Since $\mathsf{L}(A) \triangleq A + \mathsf{L}(A)$ we get maps

$$\eta^\forall : A \to \mathsf{L}(A) \qquad \mathsf{step}^\forall : \mathsf{L}(A) \to \mathsf{L}(A)$$
$$\eta^\forall a = \mathsf{inl}\, a \qquad \mathsf{step}^\forall a = \mathsf{inr}\, a$$

Lemma 1.3.19. $\mathsf{L}(A)$ *carries a monad structure with unit* $\eta^\forall$ *and multiplication* $\mu^\forall(d) \triangleq \Lambda\kappa.\mu^\kappa(d[\kappa])$*, where the Kleisli extension* $\overline{f}$ *satisfies* $\overline{f}(\mathsf{step}^\forall d) = \mathsf{step}^\forall(\overline{f}(d))$.

Proof. This is a consequence of Møgelberg and Zwart [85] and the fact that we have proven L^κ to be a monad. □

We often write $a >\!\!>\!\!= f$ for $\overline{f}(a)$ where $\overline{f}$ is the Kleisli extension.

Coinductive Convex Delay Monad Let A be clock irrelevant. As already hinted in Section 1.3, the correct monad to interpret $\mathsf{FPC}_{\oplus}$-programs with values in A is not the guarded, but the coinductive solution $\mathsf{D}^{\forall}A \triangleq \forall\kappa.\mathsf{D}^{\kappa}A$ to the domain equation induced by the functor $F_{\mathsf{D}(A)}(X) = \mathscr{D}(A+X)$.

$$\mathsf{D}^{\forall}X \simeq \mathscr{D}(X + \mathsf{D}^{\forall}X).$$

$\mathsf{D}^{\forall}A$ exists, since by Proposition 1.3.18 and Theorem 1.3.17 the functor $F_{\mathsf{D}(A)}$ commutes with clock quantification for any clock irrelevant A. Now Theorem 1.3.14 implies that $\nu(F_{\mathsf{D}(A)}) = \mathsf{D}^{\forall}A$ as required. $\mathsf{D}^{\forall}$ carries a convex algebra structure $(\delta^{\forall}, \oplus^{\forall})$, where $\delta^{\forall}x = \delta(\mathsf{inl}\,x)$ and $\oplus^{\forall}$ is just the choice operator of $\mathscr{D}$. Furthermore, there is a map $\mathsf{step}^{\forall} : \mathsf{D}^{\forall}A \to \mathsf{D}^{\forall}A$ defined by $\mathsf{step}^{\forall}x = \delta(\mathsf{inr}\,x)$.

Finally, $\mathsf{D}^{\forall}$ is a monad as the following lemma proves.

Lemma 1.3.20. *$\mathsf{D}^{\forall}$ carries a monad structure with unit $\delta^{\forall}$ and the Kleisli extension $\overline{f} : \mathsf{D}^{\forall}A \to \mathsf{D}^{\forall}B$ of any map $f : A \to \mathsf{D}^{\forall}B$ satisfies*

$$\overline{f}(\mathsf{step}^{\forall}x) = \mathsf{step}^{\forall}(\overline{f}(x)) \qquad \overline{f}(\mu \oplus_p^{\forall} \nu) = \overline{f}(\mu) \oplus_p^{\forall} \overline{f}(\nu)$$

Proof. Similar to $\mathsf{L}A$, we first show that $\mathsf{D}^{\kappa}A$ is a family of monads indexed by κ. Now [85, Lemma 16] implies the result. □

Remark 6. Whenever we write $\mathsf{D}^{\forall}A$, it is implicitly understood that A is clock irrelevant.

Coming back to the geometric distribution of Eq. (1.11), we let $\mathsf{geo}_p \triangleq \Lambda\kappa.\mathsf{geo}_p^{\kappa}$ denote the geometric process of type $\mathbb{N} \to \mathsf{D}^{\forall}(\mathbb{N})$, which satisfies

$$\begin{aligned}\mathsf{geo}_p(0) &= (\delta^{\forall}0) \oplus_p \mathsf{step}^{\forall}(\mathsf{geo}_p(1)) \\ &= (\delta^{\forall}0) \oplus_p \left(\mathsf{step}^{\forall}\Big((\delta^{\forall}1) \oplus_p \mathsf{step}^{\forall}(\mathsf{geo}_p(2))\Big)\right) \\ &= \ldots\end{aligned}$$

We can now define a map $\mathsf{PT}_n : \mathsf{D}^{\forall}A \to [0,1]$ which computes the probability of termination after n steps. Applied to the geometric distribution we for instance get

$$\mathsf{PT}_1(\mathsf{geo}_p(0)) = p + (1-p)p.$$

1.4 Implementing a flexible multimodal proof assistant

In the remainder of this introduction, we introduce the framework for general modal type theories MTT, which in particular encompasses a guarded type theory with coinductive types (as an alternative to clocked type theory). We furthermore discuss the essential ingredients to implement such a type theory.

MTT — a multimode type theory

Strictly speaking MTT is not a type theory, but, if handed a *mode theory* $\mathscr{M}$, it turns into a dependent type theory with a judgmental structure to support modal types. A mode theory is a strict 2-category that defines which modal types to consider. We refer to its objects as *modes*, and its morphisms as *modalities*.

Intuitively, every mode (m,n,o) has an independent type theory attached to it. We write $\Gamma \vdash t : A \mathbin{@} m$ for 't is of type A at mode m'. The situation is reminiscent of Kripke worlds, where truth is local to each world.

Modalities (μ,ν,ξ) are the 1-cells between modes and the glue between the (otherwise completely independent) type theories at different modes. The variables in a context Γ can be annotated by a modality, we write $x : (\mu \mid A)$ to mean the variable x of type A annotated by μ. Furthermore, modalities induce a restriction operation on variables. Such *restricted* variables can only be used if they are annotated appropriately.

By demanding that $\mathscr{M}$ is a 2-category, we ensure that modalities compose and that there is an identity modality id_m for each mode m. The judgmental structure that modalities induce ought to respect the structure of the mode theory.

Finally, the 2-cells (α,β) of $\mathscr{M}$ induce 'natural transformations' between modalities. This introduces a bunch of novel substitutions, which mediate between modal contexts. In particular, this allows us to relax the usage of μ-annotated variables. We are free to use a variable $x : (\mu \mid A)$ which is restricted by ν, if there exists a 2-cell $\alpha : \mu \longrightarrow \nu$.

We can now easily define a mode theory that extends MTT by a modality with the structure of e.g. a comonad, a monad, or an adjoint.

Example 7 (Idempotent Comonad). Consider the mode theory $\mathscr{M}$ with a single object m, a single non-identity morphism $\mu : m \longrightarrow m$ and a 2-cell $\varepsilon : \mu \longrightarrow \mathsf{id}_m$ subject to the equations $\mu \circ \mu = \mu$ and $\varepsilon \star \mu = \mu \star \varepsilon$. This description defines $\mathscr{M}$ as a 2-category with a strictly idempotent comonad μ. Instantiating MTT with this mode theory yields a modality $\langle \mu \mid - \rangle$ together with definable operations shaping $\langle \mu \mid - \rangle$ into an idempotent comonad:

$$\mathsf{extract}_A : \langle \mu \mid A \rangle \to A \qquad\qquad \mathsf{dup}_A : \langle \mu \mid A \rangle \simeq \langle \mu \mid \langle \mu \mid A \rangle \rangle$$

To define $\mathsf{extract}_A$ we crucially leverage that we can use a μ annotated variable restricted by id_m, since there is a 2-cell $\varepsilon : \mu \longrightarrow \mathsf{id}_m$. Even this simple modal type theory is quite useful; it can serve as a replacement for the experimental version of Agda [109] used to formalize a construction of univalent universes [77].

MTT includes many other rules, in particular a subtle modal elimination rule, which deviates from the usual Fitch-style modal type theories. We refer to Gratzer et al. [58] and [55] for a comprehensive and detailed introduction.

Implementing MTT Gratzer [54] proved that MTT is normalizing. However, transforming this theoretic guarantee into an implementable algorithm is no easy task.

Firstly, Gratzer [54] considers MTT to be a fully annotated generalized algebraic theory. In practice, this requires a programmer to write a lot of redundant annotations. To minimize the effort, we follow Pierce and Turner [95] who presented a *bidirectional* type checking algorithm, building on ideas pioneered by Coquand [40].

Secondly, the authors of [54] use a gluing argument to prove normalization, and while the proof is constructive and reminiscent of defunctionalized NbE [3], it is unclear how to extract an algorithm that can be implemented. Therefore, we follow the algorithm in [3] and implement a defunctionalized normalization-by-evaluation algorithm. This technique has been used successfully with modal type theories before [57, 94].

Finally, MTT is parametrized by arbitrary 2-categories and thus the decidability of type checking hinges on the decidability of the mode theory. Since there cannot be an algorithm deciding the equality of arbitrary 2-categories, we have to provide these algorithms together with the mode theory.

Restriction to preordered MTT In what follows we restrict ourselves to a smaller class of mode theories, namely those which are preordered. A mode theory is *preordered* if there is at most one 2-cell between any pair of modalities. This has two benefits:

1. Our NbE algorithm uses a presentation of variables, which is invariant under weakening. This property is disrupted by the introduction of 2-cells. Restricting to a preordered setting allows us to omit 2-cell annotations entirely and thus recover the weakening-invariant presentation of variables.

2. There is no need to annotate variables with 2-cells in the surface language. This significantly improves the usability of MTT as a programming language.

While this excludes some modal situations, it includes the important example of a guarded type theory combined with the *everything now* modality of Clouston et al. [36].

Normalization by evaluation

To implement a type checker one has to decide type equality. This is because type theories include a conversion rule, which says that if $\Gamma \vdash M : A$ and $\Gamma \vdash A = B$ then $\Gamma \vdash M : B$. Thus, validating a derivation of $\Gamma \vdash M : B$ might require an equality check. For simple type theories, type equality is often trivial, which cannot be said about

equality in a dependent setting. Here, type and term equality is entangled and deciding the equality of types is just as hard as deciding the equality of terms.

We approach the problem by singling out a special subset of *normal form* terms Nf. This allows us to reduce term equality to equality of normal forms, by defining a function $\underline{\mathbf{norm}} : \mathsf{Tm} \to \mathsf{Nf}$ which validates the following *soundness* and *completeness* requirement.

- *Soundness*: $\Gamma \vdash \underline{\mathbf{norm}}(M) = M : A$ for any term $\Gamma \vdash M : A$.
- *Completeness*: $\Gamma \vdash M = N : A$ if and only if $\underline{\mathbf{norm}}(M) = \underline{\mathbf{norm}}(N)$.

In our situation, normal form terms are precisely the terms that have no β-redeces and are η-expanded exactly once.

Example 8. The normal form of the β-redex $(\lambda x.x)0$ is 0.

Example 9. The normal form of a variable $x : \mathsf{Nat} \to \mathsf{Nat}$, is $\lambda y.xy : \mathsf{Nat} \to \mathsf{Nat}$.

Example 10. The normal form of a variable $x : (\mathsf{Nat} \to \mathsf{Nat}) \to \mathsf{Nat}$ is $\lambda f.x(\lambda y.fy)$

Ne denotes the *neutral terms*, a subclass of the normal forms. They include variables (de Bruijn Indices) and eliminators which are stuck on variables.

$$\mathsf{Nf} \supset \mathsf{Ne} \supset \mathsf{Index}$$

Example 11. For any variable x the term $x(0) \in \mathsf{Ne}$ and thus $x(0)$ is in normal form.

Representing variables While we use ordinary named variables in the surface syntax of MTT, we switch to de Bruijn indices to represent variables internally. Instead of naming variables, a *de Bruijn index* points to its binder, counting from the inside out. For instance, $\lambda x.x$ becomes $\lambda(\mathbf{q}_0)$ and an open term $x_1 : A \to B, x_2 : C \vdash \lambda(y : A).\lambda z.x_1 y$ turns into $A \to B, C \vdash \lambda(\lambda(\mathbf{q}_3(\mathbf{q}_1)))$.

Terms represented in this way are for obvious reasons invariant under the renaming of bound variables. This property is often called α-equivalence and can be annoyingly pesky if not considered carefully. Note that de Bruijn indices completely sidestep this problem, as e.g. $\lambda x.x$ and $\lambda y.y$ both become $\lambda(\mathbf{q}_0)$.

During normalization, we also use *de Bruijn levels*, which differ from indices since they count from outwards in. Here, the open term $x_1 : A \to B, x_2 : C \vdash \lambda(y : A).\lambda z.x_1 y$ becomes $A \to B, C \vdash \lambda(\lambda(\mathbf{q}_0(\mathbf{q}_2)))$. Note that if $\Gamma \vdash \mathbf{q}_k : A$ is a de Bruijn level and Γ has length n, then $\Gamma \vdash \mathbf{q}_{n-(k+1)} : A$ denotes the same variable as a de Bruijn index. Importantly, levels are invariant under weakening to the right, which allows us to define an evaluation function without ever having to shift the indices.

Normalization by Evaluation Berger and Schwichtenberg [20] discovered an algorithm to compute normal forms by first embedding and evaluating a term into a host language (in this case Scheme), and then quoting the result back to a lambda expression. Since then, there have been many different iterations of NbE. Abel [3] gives a great overview of the existing approaches.

In our situation, we define a partial evaluation function, which sends raw terms to a domain of semantic values Val. Semantic values are similar to terms, except that they exclude any term with β-redeces. Instead, we require semantic operators on the domain, which enforces that during evaluation all terms are β-reduced.

Not all β-redeces can be reduced as eliminators can get "stuck" on variables or other stuck eliminators. Thus, we define a distinguished domain D^{ne} of *neutrals*, which can be lifted to Val. The lifted semantic neutrals are then the target of the evaluation function for irreducible β-redeces.

Extension to η-equality At this point, we could quote neutrals and values back to terms and they would be in β-normal form. However, the product and the function type have η-equality, which is why normal form terms of these types should be η-expanded.

We first note, that while β-reductions can be carried out regardless of the type, η-expansions are by nature type-directed: It only makes sense to expand M to $\lambda x.Mx$ if M is a function.

Thus, we need to reify the semantic values to *normals* by annotating them with a semantic value type $\downarrow^A$. We furthermore annotate semantic neutrals with a semantic value type when they are lifted to semantic values by $\uparrow^A$. During quotation, the reification by a type that has η-equality triggers an outer η-expansion. Similarly, the lifting of semantic neutrals triggers η-expansion inside the term.

Note the deliberate choice of annotating values with *semantic value types* and not just syntactic types. This is because the type has to be β-reduced. For example, let $A(\mathsf{zero}) = \mathsf{Nat} \to \mathsf{Nat}$ and $A(\mathsf{suc}\, n) = \mathsf{Nat}$. That the term $M : A(\mathsf{zero})$ has to be η-expanded becomes only apparent after β-reducing $A(\mathsf{zero})$.

Defunctionalized NbE In the original definition of normalization-by-evaluation, functions are evaluated to actual meta-level functions. Since the domain of semantic values is untyped, this necessitates that $\mathsf{Val} \simeq (\mathsf{Val} \to \mathsf{Val})$ which gets us into the domain of *domain theory*.

It turns out that one can significantly simplify the semantic domains by replacing the meta-level functions with closures. A *closure* saves the remaining unevaluated term and the current semantic environment and suspends the computation until more information is provided.

Bidirectional type checking

As discussed before, we adapt the approach of Coquand [40] to optimize the usability of MTT as a programming language. Thus, we define the following relations simultaneously by induction

$$\Xi \vdash M \Leftarrow A \mathbin{@} m \qquad\qquad \Xi \vdash M \Rightarrow A \mathbin{@} m$$

Here, $\Xi \vdash M \Leftarrow A \mathbin{@} m$ means that we can *check* that M has type A at mode m in semantic context Ξ. Similarly, $\Xi \vdash M \Rightarrow A \mathbin{@} m$ means that we can *synthesize* the type A at mode m from term M in semantic context Ξ.

The semantic context Ξ organizes the variables currently in scope. These contexts are more complicated than usual, as they need to store environments to evaluate terms during type checking. This is needed, as we have to repeatedly check type equality for the conversion rule.

Modal annotations and restrictions additionally complicate the structure of semantic contexts. Variables are stored with their modal annotations, and restrictions $\Xi.\{\mu\}$ lazily restrict all variables in Ξ by μ.

Using this information, we perform the following checks during type checking:

- We keep track of whether a term or type can be used at a specific mode. In particular, the domain and codomain of modalities must be matched appropriately.
- We check that variables are correctly restricted. To do this, we elaborate the unique 2-cell between two modalities, interfacing with the functions provided by the mode theory. The elaboration succeeds if and only if the variable is used correctly.

1.5 Contributions and Structure

As discussed in the introduction, the contributions of this thesis can be categorized into two fields: *Programming language semantics in guarded type theory* and *implementation of a multimodal proof assistant*. In this section, I give an overview of the results of my research and close with a statement of personal contribution.

Programming language semantics in guarded type theory

We use a guarded type theory to define the operational and denotational semantics of two programming languages — FPC and $\mathsf{FPC}_{\oplus}$. By doing so, we provide the first semantics of $\mathsf{FPC}_{\oplus}$ in a *constructive type theory*, as previous work relied on classical logic for giving semantics to probabilistic programs.

Then, we define a logical relation between semantic and syntactic values and prove that its lifting is compatible with typing rules. This is sufficient to imply contextual refinement.

Even though $\mathsf{FPC}_{\oplus}$ extends FPC, there are significant conceptual differences in the approach.

A programming language with recursive types Firstly, we inductively define the standard small-step and big-step operational semantics for FPC. Then, we introduce the notion of *contextual refinement* using the big step semantics and a theory of *typed contexts*.

Equivalently, the big step semantics can be represented by using a guarded and coinductive evaluator together with a *termination relation*: If the evaluation $\mathsf{eval}(M)$ terminates to a value V, then $M \Downarrow V$. We define an interpretation function that assumes values in semantic domains. Importantly, we can prove a *soundness* theorem, which relates the denotational semantics with the operational semantics.

Using a relational lifting, we construct a logical relation that implies contextual refinement.

A probabilistic programming language with recursive types Both the operational and denotational semantics make use of a novel guarded convex delay monad, which is defined as the solution to a guarded recursive type equation. It naturally extends guarded delay monads and provides a similar interface. Thereby, many proofs are easily adapted.

In contrast to the work on FPC, we do not present small-step and big-step operational semantics by an inductive definition. Instead, we only define a guarded recursive and a coinductive evaluator. This is partly because the big-step semantics is more complicated for probabilistic programs, which are not related to values but to distributions with infinite support. Furthermore, such a semantics is not needed for the lifting of relations, which instead falls back on probabilistic *couplings*. *Contextual refinement* is then defined in terms of the coinductive evaluator and the *probability of termination*, which we can approximate for convex delay algebras.

We develop the basic constructive theory of couplings for convex delay algebras and use that to define a logical relation, relating denotational and operational semantics. It then can be shown that this relation implies contextual refinement.

In the end, we demonstrate how to use the semantics to reason about examples that combine probabilistic choice and recursion.

Conclusion and Future Work We give a constructive definition of operational and denotational semantics for call-by-value FPC. Additionally, we illustrate the utility of this approach and define a logical relation that allows us to reason about contextual refinement. Many techniques of §2 are going to be transferable to languages that extend FPC.

Furthermore, we develop a notion of (guarded) convex delay algebras and show how to use it to define and relate operational and denotational semantics for $\mathsf{FPC}_{\oplus}$ in guarded type theory. There are several paths forward from here.

Firstly, the logical relation in §3 has an undesired asymmetry to it: Consider a program M that terminates to V with probability 1, but only in the limit. Using our logical relation, we might be able to prove that M is a contextual refinement of V, but not the other way around. This is unsatisfying, and the goal is to extend the logical relation to account for approximate relational reasoning, which would allow us to prove that constant functions are refinements of their approximations.

Secondly, the present work can be extended with the account of nondeterminism in [84]. It would be interesting to compare the resulting model to the recent classically defined operationally-based logical relation in [7].

Implementing a multimode proof assistant

We contribute the flexible proof assistant `mitten`, which can be specialized to a wide range of modal type theories. We have designed normalization and type-checking algorithms for `mitten` which are motivated by recent advancements in the metatheory of MTT [54]. Finally, we construct various classical examples and show how to instantiate `mitten` to a mode theory. In particular, we define the mode theory that combines guarded recursion with the everything now modality, which renders a type theory with guarded recursive and coinductive types. At the end, we discuss further directions and related works.

Publications and statement of personal contribution

This thesis includes the following published papers and manuscripts:

- §2 is a technical report which is the result of a collaboration with Rasmus Møgelberg and Maaike Zwart. I wrote the report alone and worked out all the proofs. Maaike Zwart proved one lemma of the main theorem and Rasmus Møgelberg had an advising role, but also made many revisions.

  The introduction of the technical report is shortened since many aspects are already included in the introduction of the thesis.

- §3 is a paper currently in submission which was written with Rasmus Møgelberg, Maaike Zwart, Alejandro Aguirre and Lars Birkedal. Some technical proofs are only sketched and outsourced to the appendix. I am the first author of the paper and lead the research. In particular, I proved the soundness theorem, the compatibility lemmas, the fundamental lemma and the congruence theorem. I lead the writing of the paper and wrote a large part of it, in particular Section 3.6, Section 3.8, Section 3.9 and the respective sections in the appendix.

- §4 is a verbatim inclusion of [101]. I am the first author of the paper and conducted all the research, advised by my coauthors. I lead the writing of the paper and wrote the majority of it. Daniel Gratzer wrote Section 4.1 and Section 4.7 and revised and edited all other sections. Lars Birkedal also revised and edited the submission.

- During my internship at Airbus Cyber security I coauthored the paper "*Don't Panic! Analysing the Impact of Attacks on the Safety of Flight Management Systems*" [31] It is not part of this thesis.

Part II

Publications

Chapter 2

Technical Report: Modeling FPC in Guarded Type Theory

2.1 Introduction

Call-by-value FPC is a typed λ-calculus extended by recursive types. Its denotational and operational semantics have been extensively studied (e.g. in [100]).

Our goal is to develop operational and denotational semantics of FPC entirely in *guarded type theory*. We furthermore define the notions of contextual refinement as well as contextual equivalence, which is considered the correct notion of equality for programs. It is in general difficult to prove that programs are contextually equivalent, and therefore we show the usefulness of our approach by constructing a logical relation that implies contextual refinement.

This method has been successfully employed before by Paviotti et al. [91] as well as Møgelberg and Vezzosi [88] for different languages.

This semantics is not exactly a model of FPC, as it distinguishes contextually equivalent programs that take different numbers of computation steps. However, this approach is promising for the following reasons:

- The semantics is sound, meaning that denotationally equal terms are contextually equivalent.
- We expect that this approach can be extended to more effects than partiality, such as non-determinism or probabilistic choice.

Structure of the technical report

In what follows, we show how to use guarded recursion to prove *contextual equivalence* (also known as operational equivalence) of call-by-value FPC programs.

- In Section 2.2 we introduce the syntax and type theory of the programming language FPC. We furthermore specify four operational semantics: A small

step semantics $\rightsquigarrow$, a big step semantics $\Downarrow$, a guarded recursive evaluator eval^κ as well as a coinductive evaluator eval.

Additionally, we define a *termination relation* $\Downarrow^\forall : \mathsf{L}(A) \to A \to \mathsf{Prop}$ which relates the *total* elements of $\mathsf{L}(A)$ with elements of type A. We prove a theorem relating the big step semantics and termination predicate: If $M \Downarrow V$, then also $\mathsf{eval}(M) \Downarrow^\forall V$. Finally, we also introduce typed contexts and the notion of *contextual refinement*.

- In Section 2.3 we define guarded recursive and coinductive denotational semantics as well as interpretation functions. As before, the coinductive interpretation is directly derived from the guarded recursive one.
- In Section 2.3 we relate *evaluation* and *interpretation*.
- In Section 2.4 we show how to lift relations on values to relations on terms.
- In Section 2.5 we define a logical relation and prove structural lemmas. Finally, we establish that it implies contextual refinement (the main result).

2.2 The programming language FPC

We consider the theory FPC which extends a simple type theory with all recursive types. As a general convention, we denote type variables with capital letters X, Y, Z and term variables by x, y, z.

Remark 12. Correctly implementing variables is technical and needs to anticipate desired structural properties of the theory such as α-equivalence. We avoid this topic and assume a suitable implementation for our requirements exists.

Values are denoted by capital V, W and terms by L, M and N. All of these stand-ins might be indexed or primed if needed.

We now define inductively sets of *pre-types* PreTy, *pre-terms* PreTm and *contexts*.

$$
\begin{array}{llll}
\textit{(types)} & \sigma, \tau & ::= & 1 \mid \mathsf{Nat} \mid \sigma \to \tau \mid \sigma \times \tau \mid \sigma + \tau \mid \mu X.\tau \mid X \\
\textit{(values)} & V, W & ::= & x \mid \langle\rangle \mid \underline{n}\ (n : \mathbb{N}) \mid \mathsf{lam}\ x.M \mid \mathsf{fold}\, V \mid \langle V, W \rangle \\
 & & & \mid \mathsf{inl}\, V \mid \mathsf{inr}\, V \\
\textit{(terms)} & L, M, N & ::= & x \mid \langle\rangle \mid \underline{n}\ (n : \mathbb{N}) \mid \mathsf{suc}\, M \mid \mathsf{pred}\, N \mid \mathsf{ifz}(L, M, N) \\
 & & & \mid MN \mid \langle M, N \rangle \mid \mathsf{fst}\, M \mid \mathsf{snd}\, M \mid \mathsf{inl}\, M \mid \mathsf{inr}\, M \\
 & & & \mid \mathsf{case}(L, M, N) \mid \mathsf{fold}\, M \mid \mathsf{unfold}\, M \mid \mathsf{lam}\ x.M \\
\textit{(contexts)} & C & ::= & [\cdot] \mid CM \mid MC \mid \mathsf{lam}\ x.C \mid \mathsf{suc}\, C \mid \mathsf{pred}\, C \mid \\
 & & & \mathsf{ifz}(M, C, N) \mid \mathsf{ifz}(C, M, N) \mid \mathsf{ifz}(M, N, C) \\
 & & & \mid \mathsf{fold}\, C \mid \mathsf{unfold}\, C \mid \langle C, M \rangle \mid \langle M, C \rangle \mid \ldots \\
\textit{(ev. contexts)} & E & ::= & [\cdot] \mid EM \mid (\mathsf{lam}\ x.M)E \mid \mathsf{suc}\, E \mid \mathsf{ifz}(E, M, N) \\
 & & & \mid \mathsf{pred}\, E \mid \mathsf{inl}\, E \mid \mathsf{inr}\, E \mid \mathsf{case}(E, M, N) \mid \mathsf{fold}\, E \\
 & & & \mid \mathsf{unfold}\, E \mid \langle E, M \rangle \mid \langle V, E \rangle \mid \mathsf{fst}\, E \mid \mathsf{snd}\, E
\end{array}
$$

We come back to contexts in Section 2.2, where we present a theory for well-formed contexts and define *contextual refinement*. *Evaluation contexts* are convenient in the definition of the small-step operational semantics but also occur in Section 2.4 in the lifting of relations.

We can define a decidable predicate $\mathscr{V} : \mathsf{PreTm} \to \mathsf{bool}$ which is true if and only if a term is a *value*. Formally, $\Gamma \vdash V : \sigma$ is an abbreviation for $\Gamma \vdash M : \sigma \wedge \mathscr{V}(M)$.

Furthermore, we define a function $\mathsf{free}(M)$ which collects the free variables of a term.

Substitution on types and terms

The evaluation of terms and types is discussed in Section 2.2. There we define a Call-by-Value operational semantics for FPC. Crucially, during this process we replace variables — both type and term variables — with values and types respectively. This is a purely syntactical operation, which we can define on our pre-syntax. For types, $\sigma[\tau/X]$ denotes the *capture-free* substitution of X by τ. By capture-free, we mean that bound variables are not substituted, and free variables in τ must not be captured by binders during substitution.

$$\boxed{\sigma[\tau/X] : \mathsf{PreTy}}$$

$$\mathsf{Nat}[\tau/X] \triangleq \mathsf{Nat} \qquad (\sigma \to \sigma_1)[\tau/X] \triangleq \sigma[\tau/X] \to \sigma_1[\tau/X] \qquad \mu X.\sigma[\tau/X] \triangleq \mu X.\sigma$$

$$\mu X.\sigma[\tau/Y] \triangleq \mu X.(\sigma[\tau/Y]), \text{where } X \text{ not free in } \tau \qquad X[\tau/X] \triangleq \tau \qquad \ldots$$

For terms on the hand, we write $M[\delta]$, where $\delta = V_1/x_2, \ldots, V_n/x_n$ is a finite list of value variable pairs, to denote the *simultaenous and capture-free* substitution of the variables $x_1, \ldots, x_n$ with $V_1, \ldots, V_n$ in M. With $\mathsf{free}(\delta)$ we denote the set of free variables in $V_1, \ldots, V_n$ unioned with the set $\{x_1, \ldots x_n\}$. Practically, this just ensures that $x \notin \mathsf{free}(\delta)$ is fresh.

$$\boxed{M[\delta] : \mathsf{PreTm}}$$

$$\frac{\delta = \cdot}{x[\delta] \triangleq \mathsf{undefined}} \qquad \frac{\delta = \delta', V/x}{x[\delta] \triangleq V} \qquad \langle M, N\rangle[\delta] \triangleq \langle M[\delta], N[\delta]\rangle$$

$$\frac{x \notin \mathsf{free}(\delta)}{(\mathsf{lam}\ x.M)[\delta] \triangleq \mathsf{lam}\ x.M[\delta]}$$

$$\frac{x, y \notin \mathsf{free}(\delta)}{(\mathsf{case}(L, x.M, y.N))[V/z] \triangleq \mathsf{case}(L, x.M[V/z], y.N[V/z])} \qquad \ldots$$

Remark 13. Note that demanding $x \notin \mathsf{free}(\delta)$ is not a restriction, since we can simply rename bound variables to ensure this is the case.

Type Derivations

FPC has general recursive types $\mu X.\tau$. Thus, unlike in a simple type theory, we cannot specify well-formed types by grammar. Hence, we define a relation $\vdash$ which tells us whether a type is well-formed in a given type variable context Φ. To avoid delving too deep into implementation details, we assume that type contexts are finite sets of type variables. Furthermore, there is an infinite pool of fresh variable names.

Definition 2.2.1 (Type Contexts)**.** We use *type variable contexts* Θ to denote the sets of type variables X, Y, Z. We write $X \in \Theta$ to denote membership. If X is fresh, Θ, X adds X to Θ and iteratively, for disjoint sets Θ and Θ', the set Θ, Θ' denotes their union.

Thus, for any Θ we get the following rules

$$\frac{X \in \Theta}{\Theta \vdash X} \qquad \frac{}{\Theta \vdash \mathsf{Nat}} \qquad \frac{}{\Theta \vdash 1} \qquad \frac{\Theta \vdash \sigma \qquad \Theta \vdash \tau}{\Theta \vdash \sigma \to \tau} \qquad \frac{\Theta \vdash \sigma \qquad \Theta \vdash \tau}{\Theta \vdash \sigma \times \tau}$$

$$\frac{\Theta \vdash \sigma \qquad \Theta \vdash \tau}{\Theta \vdash \sigma + \tau} \qquad \frac{\Theta, X \vdash \tau}{\Theta \vdash \mu X.\tau}$$

Figure 2.1: types

Type Theory

Open terms can only be well-typed if a *typing context* provides meaning to its free variables.

As for type derivations, we avoid discussing implementation details of contexts and variables and instead presuppose that Γ is a finite set and that there is a decidable relation $(x : \sigma) \in \Gamma$. Using this, we may define $x \in \Gamma \triangleq \exists \sigma.(x : \sigma) \in \Gamma$. Note that this is also decidable since Γ is finite.

We write $\Gamma, x : \sigma \triangleq \Gamma \cup \{x : \sigma\}$ and iteratively $\Gamma, \Delta \triangleq \Gamma \cup \Delta$.

Definition 2.2.2 (Well-formed Context)**.** We define a predicate $\vdash \Gamma$ to denote that a *typing context* Γ is well-formed

$$\frac{}{\vdash \cdot} \qquad \frac{\vdash \Gamma \qquad \vdash \sigma \qquad x \notin \Gamma}{\vdash \Gamma, (x : \sigma)}$$

Now we can define the typing relation $\Gamma \vdash M : \sigma$ for FPC.

$$\frac{x : \sigma \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x : \sigma} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{n \in \mathbb{N} \qquad \vdash \Gamma}{\Gamma \vdash \underline{n} : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\, M : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}\, M : \mathsf{Nat}} \qquad \frac{\Gamma \vdash L : \mathsf{Nat} \qquad \Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathsf{ifz}(L, M, N) : \sigma}$$

$$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathsf{inl}\, M : \sigma + \tau} \qquad \frac{\Gamma \vdash M : \tau}{\Gamma \vdash \mathsf{inr}\, M : \sigma + \tau} \qquad \frac{\Gamma \vdash M : \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash \langle M, N \rangle : \sigma \times \tau}$$

$$\frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathsf{fst}\, M : \sigma} \qquad \frac{\Gamma \vdash M : \sigma \times \tau}{\Gamma \vdash \mathsf{snd}\, M : \tau}$$

$$\frac{\Gamma \vdash L : \sigma + \sigma' \qquad \Gamma, (x : \sigma) \vdash M : \tau \qquad \Gamma, (y : \sigma') \vdash N : \tau}{\Gamma \vdash \mathsf{case}(L, x.M, y.N) : \sigma}$$

$$\frac{\Gamma, (x : \sigma) \vdash M : \tau}{\Gamma \vdash \mathsf{lam}\, x.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash N : \sigma \to \tau \qquad \Gamma \vdash M : \sigma}{\Gamma \vdash NM : \tau} \qquad \frac{\Gamma \vdash M : \tau[\mu X.\tau / X]}{\Gamma \vdash \mathsf{fold}\, M : \mu X.\tau}$$

$$\frac{\Gamma \vdash M : \mu X.\tau}{\Gamma \vdash \mathsf{unfold}\, M : \tau[\mu X.\tau / X]}$$

We may write $\vdash M : \sigma$ for $\cdot \vdash M : \sigma$.

Remark 14. A typing relation $\vdash$ is usually defined as an inductive family of propositions. While the HIT scheme of Cohen et al. [37] supports them, Kristensen et al. [72] did not consider the indexing of HITs for complexity reasons. They conjecture the scheme can be extended though. As a temporary solution, however, we consider the syntax to be fully annotated. Consequently, type checking becomes decidable and $\Gamma \vdash M : \sigma$ is a notation for the recursively defined function

$$(- \vdash - : -) : \mathsf{TypingContext} \to \mathsf{PreTm} \to \mathsf{PreTy} \to \mathsf{bool}$$

Definition 2.2.3. We define the sets of all well-formed types, terms and values respectively.

$$\mathsf{Ty} \triangleq \Sigma(\sigma \in \mathsf{PreTy})(\vdash \sigma) \qquad \mathsf{Tm}_\sigma^\Gamma \triangleq \Sigma(M \in \mathsf{PreTm})(\Gamma \vdash M : \sigma)$$

$$\mathsf{Val}_\sigma^\Gamma \triangleq \Sigma(V \in \mathsf{PreVal})(\Gamma \vdash V : \sigma).$$

With Tm_σ and Val_σ, we denote the special cases where Γ is the empty context.

For closed value types it is easy to prove the following correspondence by induction on the typing derivation.

Lemma 2.2.4 (Canonical Form lemma)**.**

$$\begin{aligned} \mathsf{Val}_{\sigma \times \tau} &\simeq \mathsf{Val}_\sigma \times \mathsf{Val}_\tau & \mathsf{Val}_{\sigma + \tau} &\simeq \mathsf{Val}_\sigma + \mathsf{Val}_\tau \\ \mathsf{Val}_{\sigma \to \tau} &\simeq \Sigma(\mathsf{lam}\, x.M : \mathsf{PreVal})(\cdot \vdash \mathsf{lam}\, x.M : \sigma \to \tau) & \mathsf{Val}_{\mathsf{Nat}} &\simeq \mathbb{N} \\ \mathsf{Val}_{\mu X.\tau} &\simeq \Sigma(\mathsf{fold}\, M : \mathsf{PreVal})(\cdot \vdash \mathsf{fold}\, M : \mu X.\tau) & \mathsf{Val}_1 &\simeq 1 \end{aligned}$$

Proof. The proof is a straightforward induction. □

This lemma justifies that we overload expressions and use the meta-level functions $\mathbf{suc}, \mathbf{pred} : \mathbb{N} \to \mathbb{N}$ as well as $\mathsf{pr}_1 : \mathsf{Val}_{\sigma\times\tau} \to \mathsf{Val}_\sigma$ and $\mathsf{pr}_2 : \mathsf{Val}_{\sigma\times\tau} \to \mathsf{Val}_\tau$ synonimously with FPC value constructors.

$$\mathsf{suc}\,(\underline{n}) = \mathbf{suc}(n) \qquad \mathsf{fst}\,V = \mathsf{pr}_1 V$$
$$\mathsf{pred}\,(\underline{n}) = \mathbf{pred}(n) \qquad \mathsf{snd}\,V = \mathsf{pr}_2 V$$

We also define functions out of $\mathsf{Val}_{\sigma+\tau}$, $\mathsf{Val}_{\sigma\to\tau}$ and $\mathsf{Val}_{\mu\mathsf{X}.\tau}$ by matching on cases, such as

$$\begin{cases} \mathsf{inl}\,V \mapsto a_1 \\ \mathsf{inr}\,V \mapsto a_2 \end{cases} : \mathsf{Val}_{\sigma+\tau} \to A$$
$$\lambda(\mathsf{lam}\ x.M).a : \mathsf{Val}_{\sigma\to\tau} \to A \qquad \lambda(\mathsf{fold}\,V).a : \mathsf{Val}_{\mu\mathsf{X}.\tau} \to A.$$

This is justified, as all elements of $\mathsf{Val}_{\sigma\to\tau}$ and $\mathsf{Val}_{\mu\mathsf{X}.\tau}$ are of the form $\mathsf{lam}\ x.M$ and $\mathsf{fold}\,M$ respectively.

Substitution Lemma We introduce a notion of *well-formed* substitution which preserves well-typedness

Definition 2.2.5 (Well-formed Subsitution)**.** A substitution $\delta : \mathsf{Sub}(\Gamma;\Delta)$ is a finite list of value and variable pairs, which can be defined inductively as follows:

$$\frac{}{\cdot : \mathsf{Sub}(\Gamma;\cdot)} \qquad \frac{\delta : \mathsf{Sub}(\Gamma;\Delta) \qquad \Gamma \vdash V : \sigma \qquad x \notin \Delta}{\delta, V/x : \mathsf{Sub}(\Gamma;\Delta,(x:\sigma))}$$

With $\delta(x)$ we denote the value that is substituted for x.

There is an obvious identity substitution $\mathsf{id} : \mathsf{Sub}(\Gamma;\Gamma)$ for any context Γ. We simply write $M[V/x]$ instead of $M[\mathsf{id}, V/x]$.

Lemma 2.2.6 (Substitution Lemma)**.** *If* $\Delta \vdash M : \tau$ *and* $\delta : \mathsf{Sub}(\Gamma;\Delta)$*, then* $\Gamma \vdash M[\delta] : \tau$.

Proof. The proof proceeds by induction on derivations.

Case: $\Delta \vdash \mathsf{lam}\ x.M : \sigma \to \tau$

We have that $\Delta, x:\sigma \vdash M : \tau$ and $(\delta, x/x) : \Gamma, x:\sigma \vdash \Delta, x:\sigma$. Thus, by the induction hypothesis, we get $\Gamma, x:\sigma \vdash M[\delta, x/x] : \tau$. This however directly implies that $\Gamma \vdash \mathsf{lam}\ x.(M[\delta]) : \sigma \to \tau$

Case: $\Delta \vdash x : \sigma$

Then $(x:\sigma) \in \Delta$ and since $\delta : \Gamma \to \Delta$, there exist a V, such that $V/x \in \delta$. But then $\Gamma \vdash V : \sigma$ and the claim follows.

Case: $\Delta \vdash MN : \tau$

We have $\Delta \vdash M : \sigma \to \tau$ and $\Delta \vdash N : \sigma$ and thus the induction hypothesis gives us By induction hypothesis we get that $\Gamma \vdash M[\delta] : \sigma \to \tau$ and $\Gamma \vdash N[\delta] : \sigma$. Thus, $\Gamma \vdash (M[\delta])N[\delta] : \tau$ and the claim follows.

The remaining cases are similar □

Operational Semantics

In Fig. 2.2 we define the small-step operational semantics for this theory by induction on terms.

$$(\mathsf{lam}\, x.M)V \rightsquigarrow M[V/x] \qquad \mathsf{suc}\,\underline{n} \rightsquigarrow \underline{n+1} \qquad \mathsf{pred}\,\underline{n} \rightsquigarrow \mathbf{max}(\underline{0},\underline{n-1})$$

$$\mathsf{ifz}(\underline{0},M,N) \rightsquigarrow M \qquad \mathsf{ifz}(\underline{n+1},M,N) \rightsquigarrow N \qquad \mathsf{case}(\mathsf{inl}\,V,x.M,y.N) \rightsquigarrow M[V/x]$$

$$\mathsf{case}(\mathsf{inr}\,V,x.M,y.N) \rightsquigarrow N[V/y] \qquad \mathsf{fst}\,\langle V,W\rangle \rightsquigarrow V \qquad \mathsf{snd}\,\langle V,W\rangle \rightsquigarrow W$$

$$\mathsf{unfold}\,\mathsf{fold}\,V \rightsquigarrow V \qquad \frac{M \rightsquigarrow N}{E[M] \rightsquigarrow E[N]}$$

Figure 2.2: Small Step Operational Semantics for FPC

We form the reflexive, transitive closure $\rightsquigarrow^*$

$$\frac{}{M \rightsquigarrow^* M} \qquad \frac{M \rightsquigarrow^* M_1 \qquad M_1 \rightsquigarrow M_2}{M \rightsquigarrow^* M_2}$$

The big step operational semantics is defined in Fig. 2.3.

$$\boxed{M \Downarrow V}$$

$$\frac{M = V}{M \Downarrow V} \qquad \frac{M \Downarrow \underline{n}}{\mathsf{suc}\,M \Downarrow \underline{n+1}} \qquad \frac{M \Downarrow \underline{n}}{\mathsf{pred}\,M \Downarrow \mathbf{max}(0,n-1)} \qquad \frac{L \Downarrow \underline{0} \qquad M \Downarrow V}{\mathsf{ifz}(L,M,N) \Downarrow V}$$

$$\frac{L \Downarrow \underline{n+1} \qquad N \Downarrow V}{\mathsf{ifz}(L,M,N) \Downarrow V} \qquad \frac{M \Downarrow V}{\mathsf{inl}\,M \Downarrow \mathsf{inl}\,V} \qquad \frac{M \Downarrow V}{\mathsf{inr}\,M \Downarrow \mathsf{inr}\,V}$$

$$\frac{L \Downarrow \mathsf{inl}\,V \qquad M[V/x] \Downarrow W}{\mathsf{case}(L,x.M,y.N) \Downarrow W} \qquad \frac{L \Downarrow \mathsf{inr}\,V \qquad N[V/y] \Downarrow W}{\mathsf{case}(L,x.M,y.N) \Downarrow W} \qquad \frac{M \Downarrow V \qquad N \Downarrow W}{\langle M,N\rangle \Downarrow \langle V,W\rangle}$$

$$\frac{M \Downarrow \langle V,W\rangle}{\mathsf{fst}\,M \Downarrow V} \qquad \frac{M \Downarrow \langle V,W\rangle}{\mathsf{snd}\,M \Downarrow W} \qquad \frac{M \Downarrow \mathsf{lam}\, x.M' \qquad N \Downarrow V' \qquad M'[V'/x] \Downarrow V}{MN \Downarrow V}$$

$$\frac{M \Downarrow V}{\mathsf{fold}\,M \Downarrow \mathsf{fold}\,V} \qquad \frac{M \Downarrow \mathsf{fold}\,V}{\mathsf{unfold}\,M \Downarrow V}$$

Figure 2.3: big step semantics

The following lemmas prove that the small and big step semantics are compatible.

Lemma 2.2.7. *If $M \rightsquigarrow M'$ then $M \Downarrow V \leftrightarrow M' \Downarrow V$*

Proof. The proof proceeds by induction on $M \rightsquigarrow M'$. We also need to distinguish all possible cases for evaluation contexts E.

For instance, let $(\mathsf{lam}\ x.M)V \rightsquigarrow M[V]$. If furthermore $(\mathsf{lam}\ x.M)V \Downarrow W$, then by induction on $\Downarrow$ we get that $M[V/x] \Downarrow W$. If on the other hand, we have that $M[V/x] \Downarrow W$, then clearly $(\mathsf{lam}\ x.M)V \Downarrow W$ by the definition of $\Downarrow$.

As another example, let $E = [-]N$ and $E[M] \rightsquigarrow E[M']$. By definition of $\rightsquigarrow$, we have that $M \rightsquigarrow M'$. Now if $MN \Downarrow V$, it must be the case that $M \Downarrow \mathsf{lam}\ x.L$, $N \Downarrow W$ and $L[W/x] \Downarrow V$. The induction hypothesis implies that $M \Downarrow \mathsf{lam}\ x.L \leftrightarrow M' \Downarrow \mathsf{lam}\ x.L$, and thus the claim follows

The remaining cases follow similarly. □

This lemma extends to the transitive closure of the small-step operational semantics.

Corollary 2.2.8. *If $M \rightsquigarrow^* M'$ then $M \Downarrow V \leftrightarrow M' \Downarrow V$*

Lemma 2.2.9. *If $M \Downarrow V$, then $M \rightsquigarrow^* V$.*

Proof. The proof is by induction on $\Downarrow$ and the fact that $\rightsquigarrow^*$ is transitive. □

Remark 15. The big-step semantics as well as the reflexive transitive closure $\rightsquigarrow^*$ are inductive families of propositions, which the higher inductive type scheme in Kristensen et al. [72] does not include yet. As a temporary solution, we note that both of these families of types can be encoded by defining a *fuelled* version using induction on natural numbers. For example, let

$$\begin{aligned}
\Downarrow^n &: \mathsf{Tm}_\sigma \to \mathsf{Val}_\sigma \to \mathbb{N} \to \mathsf{Prop} \\
M \Downarrow^0 V &\triangleq M = V \\
MN \Downarrow^{n+1} V &\triangleq (M \Downarrow^n \mathsf{lam}\ x.M') \wedge (N \Downarrow^n V) \wedge (M'[V'/x] \Downarrow^n V) \\
\mathsf{unfold}\, M \Downarrow^{n+1} V &\triangleq M \Downarrow^n \mathsf{fold}\, V \\
&\vdots
\end{aligned}$$

We can now define $M \Downarrow V \triangleq \exists n.M \Downarrow^n V$

Evaluation

In Fig. 2.4 we define an *evaluator* $\mathsf{eval}^\kappa : \{\sigma : \mathsf{Ty}\} \to \mathsf{Tm}_\sigma \to \mathsf{L}^\kappa(\mathsf{Val}_\sigma)$, which evaluates a term according to the operational semantics specified in Section 2.2. This function cannot be defined inductively, as the recursive calls for *function application* and *coproduct elimination* would not be well-founded due to the substitution taking place. Thus, it is defined by guarded recursion. As such, the recursive calls are guarded by $\triangleright^\kappa$, which is why they must occur under tick like so:

$$\mathsf{name} \triangleq \mathsf{step}^\kappa(\lambda(\alpha : \kappa).f(\mathsf{name}))$$

$$
\begin{aligned}
\mathsf{eval}^\kappa(V) &\triangleq \eta^\kappa V \\
\mathsf{eval}^\kappa(\mathsf{pred}\,M) &\triangleq \mathsf{L}^\kappa(\mathbf{pred})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{suc}\,M) &\triangleq \mathsf{L}^\kappa(\mathbf{suc})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{inl}\,M) &\triangleq \mathsf{L}^\kappa(\mathbf{inl})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{inr}\,M) &\triangleq \mathsf{L}^\kappa(\mathbf{inr})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{ifz}(L,M,N)) &\triangleq \mathsf{eval}^\kappa L >\!\!>\!\!=^\kappa \begin{cases} \underline{0} \mapsto \mathsf{eval}^\kappa(M) \\ \underline{n+1} \mapsto \mathsf{eval}^\kappa(M) \end{cases} \\
\mathsf{eval}^\kappa(\mathsf{fst}\,M) &\triangleq (\mathsf{eval}^\kappa M) >\!\!>\!\!=^\kappa \lambda\langle V,W\rangle.\eta^\kappa V \\
\mathsf{eval}^\kappa(\mathsf{snd}\,M) &\triangleq (\mathsf{eval}^\kappa M) >\!\!>\!\!=^\kappa \lambda\langle V,W\rangle.\eta^\kappa W \\
\mathsf{eval}^\kappa\langle M,N\rangle &\triangleq \mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda V. \\
&\qquad \mathsf{eval}^\kappa N >\!\!>\!\!=^\kappa \lambda W.\eta^\kappa\langle V,W\rangle \\
\mathsf{eval}^\kappa(\mathsf{case}(L,x.M,y.N)) &\triangleq \mathsf{eval}^\kappa(L) \\
&\qquad >\!\!>\!\!=^\kappa \begin{cases} \mathsf{inl}\,V \mapsto \mathsf{step}^\kappa(\lambda(\alpha:\kappa).\mathsf{eval}^\kappa(M[V/x])) \\ \mathsf{inr}\,V \mapsto \mathsf{step}^\kappa(\lambda(\alpha:\kappa).\mathsf{eval}^\kappa(N[V/y])) \end{cases} \\
\mathsf{eval}^\kappa(MN) &\triangleq \mathsf{eval}^\kappa(M) >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\ x.M').\ \mathsf{eval}^\kappa(N) \\
&\qquad >\!\!>\!\!=^\kappa \lambda V.\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\mathsf{eval}^\kappa(M'[V/x])) \\
\mathsf{eval}^\kappa(\mathsf{fold}\,M) &\triangleq \mathsf{L}^\kappa(\mathsf{fold})(\mathsf{eval}^\kappa(M)) \\
\mathsf{eval}^\kappa(\mathsf{unfold}\,M) &\triangleq \mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\,V).\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\eta^\kappa V)
\end{aligned}
$$

Figure 2.4: The evaluation function eval^κ.

We often write $\mathsf{eval}^\kappa M$ instead of $\mathsf{eval}^\kappa(\cdot \vdash M : \sigma)$.

Remark 16. The evaluation of $\mathsf{unfold}\,M$ does not require guarded recursion. However, it is necessary to add a step^κ to match the steps of evaluation and *interpretation* (defined in Fig. 2.6) for proving that the semantics is operationally sound (Section 2.3).

Example 17. We define

$$
\begin{aligned}
&\cdot \vdash \mathsf{Y} : ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau \\
&\mathsf{Y} \triangleq\ \mathsf{lam}\ f.\mathsf{lam}\ z.e_f(\mathsf{fold}\,e_f)z, \text{ where} \\
&\qquad e_f : (\mu X.(X \to \sigma \to \tau)) \to \sigma \to \tau \\
&\qquad e_f \triangleq \mathsf{lam}\ y.\mathsf{let}\ y' = \mathsf{unfold}\,y\ \mathsf{in}\ f(\mathsf{lam}\ x.y'yx)
\end{aligned}
$$

Here, we have that

1. $y : \mu X.(X \to \sigma \to \tau)$

2. $y' : (\mu X.(X \to \sigma \to \tau)) \to \sigma \to \tau$
3. $\mathsf{lam}\, x.y'yx : \sigma \to \tau$

Then, for any values $\cdot \vdash f : (\sigma \to \tau) \to \sigma \to \tau$ and $\cdot \vdash V : \sigma$ the following is true

- $\mathsf{eval}^\kappa((\mathsf{Y} f)(V)) = (\Delta^\kappa)^4(\mathsf{eval}^\kappa((f(\mathsf{Y} f))(V)))$ where $\Delta^\kappa \triangleq (\mathsf{step}^\kappa \circ \mathsf{next}^\kappa)$.
- $(\mathsf{Y} f)(V) \rightsquigarrow^* (f(\lambda z.e_f(\mathsf{fold}\, e_f)z))(V)$
- $(f(\mathsf{Y} f))(V) \rightsquigarrow^* (f(\lambda z.e_f(\mathsf{fold}\, e_f)z))(V)$

Proof. Let

$$E_V \triangleq \lambda(\mathsf{lam}\, x.M).\mathsf{step}^\kappa(\lambda(\alpha : \kappa).\mathsf{eval}^\kappa(M[V/x]))$$

Note that

$$\mathsf{eval}^\kappa(\mathsf{Y}\varphi) = \Delta^\kappa(\mathsf{eval}^\kappa(\mathsf{lam}\, x.(e_\varphi(\mathsf{fold}\, e_\varphi))x))$$

and

$$\mathsf{eval}^\kappa(e_\varphi(\mathsf{fold}\, e_\varphi)) = (\Delta^\kappa)^3(\mathsf{eval}^\kappa(\varphi(\mathsf{lam}\, x.(e_\varphi(\mathsf{fold}\, e_\varphi)x)))$$

Therefore,

$$\begin{aligned}
&\mathsf{eval}^\kappa((\mathsf{Y}\varphi)(V)) \\
&= (\Delta^\kappa)^2(\mathsf{eval}^\kappa(e_\varphi(\mathsf{fold}\, e_\varphi))V)) \\
&= (\Delta^\kappa)^5(\mathsf{eval}^\kappa(\varphi(\mathsf{lam}\, x.(e_\varphi(\mathsf{fold}\, e_\varphi)x)) >>=^\kappa E_V) \\
&= (\Delta^\kappa)^5(\delta^\kappa(\varphi) >>=^\kappa \lambda(\mathsf{lam}\, Z.P).\mathsf{eval}^\kappa(\mathsf{lam}\, x.(e_\varphi(\mathsf{fold}\, e_\varphi)x)) \\
&\qquad >>=^\kappa \lambda U.\Delta^\kappa(\mathsf{eval}^\kappa(P[U/z]) >>=^\kappa E_V)) \\
&= (\Delta^\kappa)^4(\delta^\kappa(\varphi) >>=^\kappa \lambda(\mathsf{lam}\, Z.P).\Delta^\kappa.\mathsf{eval}^\kappa(\mathsf{lam}\, x.(e_\varphi(\mathsf{fold}\, e_\varphi)x)) \\
&\qquad >>=^\kappa \lambda U.\Delta^\kappa(\mathsf{eval}^\kappa(P[U/z]) >>=^\kappa E_V)) \\
&= (\Delta^\kappa)^4(\delta^\kappa(\varphi) >>=^\kappa \lambda(\mathsf{lam}\, Z.P).\mathsf{eval}^\kappa(\mathsf{Y}\varphi) \\
&\qquad >>=^\kappa \lambda U.\Delta^\kappa(\mathsf{eval}^\kappa(P[U/z]) >>=^\kappa E_V)) \\
&= (\Delta^\kappa)^4(\mathsf{eval}^\kappa((\varphi(\mathsf{Y}\varphi))) >>=^\kappa E_V) \\
&= (\Delta^\kappa)^4(\mathsf{eval}^\kappa((\varphi(\mathsf{Y}\varphi))(V)))
\end{aligned}$$

The remaining statements follow by directly evaluating the term using the rules of Fig. 2.2.

□

Coinductive Evaluator The corresponding evaluator $\mathsf{eval} : \{\sigma : \mathsf{Ty}\} \to \mathsf{Tm}_\sigma \to \mathsf{L}(\mathsf{Val}_\sigma)$ which maps terms to the (coinductive) partiality monad can be defined by

$$\mathsf{eval}(M) \triangleq \Lambda\kappa.\mathsf{eval}^\kappa M.$$

This is well-defined, since $\forall\kappa$ is an applicative functor, and furthermore by Theorem 1.3.17 $\mathsf{Tm}_\Gamma\sigma$ is clock-irrelevant for any Γ and σ.

A termination relation for total computations

Recall that for any clock irrelevant type A, the type $\mathsf{L}(A)$ denotes partial elements of A. We define a relation $m \Downarrow^{\forall} a$, which intuitively says that a partial element $m : \mathsf{L}(A)$ is actually *total* and — modulo a finite number of $\mathsf{step}^{\forall}$ — equal to $a : A$.

Note, that such a predicate would not be well-typed on $\mathsf{L}^{\kappa}(A)$. If $\mathsf{step}^{\kappa}(d) : \mathsf{L}^{\kappa}(A)$, then $d : \triangleright^{\kappa}\mathsf{L}^{\kappa}(A)$ and therefore the recursive call needs to occur under tick.

For $\mathsf{step}^{\forall}(d) : \mathsf{L}(A)$, on the other hand, we have that $d : \mathsf{L}(A)$.

$$\boxed{\Gamma, (m : \mathsf{L}(A)), (a : A) \vdash m \Downarrow^{\forall} a : \mathsf{Prop}}$$

$$\frac{}{\Gamma, (a:A) \vdash \eta^{\forall}(a) \Downarrow^{\forall} a} \qquad \frac{\Gamma \vdash m \Downarrow^{\forall} a}{\Gamma \vdash \mathsf{step}^{\forall}(m) \Downarrow^{\forall} a}$$

Remark 18. This relation is an inductive family of propositions and can be encoded similarly to the big step operational semantics (Remark 15).

As our first application of the termination predicate, consider the partial evaluation of a term $\mathsf{eval}(M) : \mathsf{L}(\mathsf{Val}_{\sigma})$, which — if $M \Downarrow V$ — should compute to a value V (potentially nested under a finite number of steps $\mathsf{step}^{\forall}$). And indeed, in Lemma 2.2.12 we show that if $M \Downarrow V$, then also $\mathsf{eval}(M) \Downarrow^{\forall} V$.

Lemma 2.2.10 (Bind Lemma)**.** *Let $m : \mathsf{L}(A)$ and $f : A \to \mathsf{L}(B)$. Then $m \Downarrow^{\forall} a$ and $fa \Downarrow^{\forall} b$ imply that $(m >>= f) \Downarrow^{\forall} b$*

Proof. We proceed by induction on $m \Downarrow^{\forall} a$.

Case: $\eta^{\forall}(a) \Downarrow^{\forall} a$

Then by defintion $(m >>= f) \Downarrow^{\forall} b = fa \Downarrow^{\forall} b$ and thus there is nothing to show.

Case: $\mathsf{step}^{\forall}(m) \Downarrow^{\forall} a$

We have that $\mathsf{step}^{\forall}(m) \Downarrow^{\forall} a = m \Downarrow^{\forall} a$ and thus we can apply the induction hypothesis and get that $m >>= f \Downarrow^{\forall} b$. We now have that

$$\begin{aligned}(m >>= f) \Downarrow^{\forall} b &= (\mathsf{step}^{\forall}(m >>= f)) \Downarrow^{\forall} b \\ &= ((\mathsf{step}^{\forall} m) >>= f) \Downarrow^{\forall} b\end{aligned}$$

and thus the claim follows. □

Corollary 2.2.11. *Let $m : \mathsf{L}(A)$ and $f : A \to B$. Then $m \Downarrow^{\forall} a$ implies $\mathsf{L}(f)(m) \Downarrow^{\forall} f(a)$*

Proof. We have that $\eta^{\forall} \circ f : A \to \mathsf{L}(B)$ and by definition $\eta^{\forall}(f(a)) \Downarrow^{\forall} f(a)$ is true. Now the Lemma 2.2.10 implies that $m >>= \eta^{\forall} \circ f \Downarrow^{\forall} f(a)$ which is by definition $\mathsf{L}(f)(m) \Downarrow^{\forall} f(a)$. □

Lemma 2.2.12. *For any well-typed closed term $\cdot \vdash M : \sigma$ we have that*

$$M \Downarrow V \to \mathsf{eval}(M) \Downarrow^{\forall} V$$

Proof. We proceed by induction on $M \Downarrow V$.

Case: $\mathsf{case}(L, x.M, y.N) \Downarrow V$

There are two subcases to consider:

1. $L \Downarrow \mathsf{inl}\,(W)$ and $M[W/x] \Downarrow V$
2. $L \Downarrow \mathsf{inr}\,(W)$ and $N[W/x] \Downarrow V$.

Case: $L \Downarrow \mathsf{inl}\,W$ and $M[W/x] \Downarrow V$

By induction hypothesis it follows that $\mathsf{eval}(L) \Downarrow^{\forall} \mathsf{inl}\,(W)$ and $\mathsf{eval}(M[W/x]) \Downarrow^{\forall} V$. Furthermore, unfolding the definition of eval gives

$$\mathsf{eval}(\mathsf{case}(L, x.M, y.N)) = \mathsf{eval}(L) >\!\!>= \begin{cases} \mathsf{inl}\,W' \mapsto \mathsf{step}^{\forall}(\mathsf{eval}(M[W'/x])) \\ \mathsf{inr}\,W' \mapsto \mathsf{step}^{\forall}(\mathsf{eval}(N[W'/y])) \end{cases}.$$

Note that $\mathsf{eval}(\mathsf{case}(\mathsf{inl}\,W', x.M, y.N)) = \mathsf{step}^{\forall}(\mathsf{eval}(M[W'/x]))$ and therefore we can use Lemma 2.2.10 to show $\mathsf{eval}(\mathsf{case}(L, x.M, y.N)) \Downarrow^{\forall} V$.

We apply it to $\mathsf{eval}(L) \Downarrow^{\forall} \mathsf{inl}\,(W)$ and $\lambda(\mathsf{inl}\,W').\mathsf{step}^{\forall}(\mathsf{eval}(M[W'/x]))$. To do so, it remains to be shown that $\mathsf{step}^{\forall}(\mathsf{eval}(M[W'/x])) \Downarrow^{\forall} V$, which is however immediate since $\mathsf{eval}(M[W'/x]) \Downarrow^{\forall} V$ by the inductive hypothesis.

Case: $L \Downarrow \mathsf{inr}\,W$ and $N[W/x] \Downarrow V$

This case is completely analogous.

Case: $MN \Downarrow V$

By definition we have that $M \Downarrow \mathsf{lam}\ x.M'$, $N \Downarrow W$ and $M'[W/x] \Downarrow V$. After applying the induction hypothesis we get

1. $\mathsf{eval}(M) \Downarrow^{\forall} \mathsf{lam}\ x.M'$
2. $\mathsf{eval}(N) \Downarrow^{\forall} W$
3. $\mathsf{eval}(M'[W/x]) \Downarrow^{\forall} V$

We have that $\mathsf{eval}(MN)$ is equal to

$$\mathsf{eval}(M) >\!\!>= \Big(\lambda(\mathsf{lam}\ x.M').\mathsf{eval}(N) >\!\!>= \lambda W.\mathsf{step}^{\forall}(\mathsf{eval}(M'[W/x]))\Big)$$

and thus we have to apply Lemma 2.2.10 for

$$\lambda(\mathsf{lam}\ x.M').\mathsf{eval}(N) >\!\!>= \lambda W.\mathsf{step}^{\forall}(\mathsf{eval}(M'[W/x])).$$

To do this, we need to check that $\mathsf{eval}(M) \Downarrow^{\forall} \mathsf{lam}\ x.M'$ — which follows from the induction hypothesis — and that

$$\mathsf{eval}(N) >\!\!>= \lambda W.\mathsf{step}^{\forall}(\mathsf{eval}(M'[W/x])) \Downarrow^{\forall} V.$$

For the latter we apply Lemma 2.2.10 to the function $\lambda W.\mathsf{step}^{\forall}(\mathsf{eval}(M'[W/x]))$: This is possible, since by the induction hypothesis both $\mathsf{eval}(N) \Downarrow^{\forall} W$ and $\mathsf{eval}(M'[W/x]) \Downarrow^{\forall} V$ are true and the claim follows.

Case: $\mathsf{unfold}\, M \Downarrow V$
By induction hypothesis we have that $\mathsf{eval}(M) \Downarrow^{\forall} \mathsf{fold}\, V$. We have that

$$\mathsf{eval}(\mathsf{unfold}\, M) = \left(\mathsf{eval}(M) >>= \lambda(\mathsf{fold}\, V).\mathsf{step}^{\forall}(\eta^{\forall}(V))\right)$$

and thus to show that $\mathsf{unfold}\, M \Downarrow^{\forall} V$, we use Lemma 2.2.10 for the function

$$\lambda(\mathsf{fold}\, V).\mathsf{step}^{\forall}(\eta^{\forall}(V)).$$

It thus suffices to show that $\mathsf{step}^{\forall}(\eta^{\forall}(V)) \Downarrow^{\forall} V$. This follows from the fact that $\mathsf{step}^{\forall}(\eta^{\forall}(V)) \Downarrow^{\forall} V = \eta^{\forall}(V) \Downarrow^{\forall} V$.

Case: $\mathsf{fold}\, M \Downarrow \mathsf{fold}\, V$ By induction hypothesis we get that $\mathsf{eval}(M) \Downarrow^{\forall} V$ and since

$$\mathsf{eval}(\mathsf{fold}\, M) = \mathsf{L}(\mathsf{fold})(\mathsf{eval}(M))$$

we use Corollary 2.2.11 for the function $\mathsf{fold} : \mathsf{Val}_{\sigma[\mu\mathsf{X}.\sigma/\mathsf{X}]} \to \mathsf{Val}_{\mu\mathsf{X}.\sigma}$. It thus suffices to show that $\mathsf{eval}(M) \Downarrow^{\forall} V$, which follows from the induction hypothesis.

The remaining cases are similar. □

Typed Contexts

While the operational semantics induces an equational theory on closed terms, this approach comes with two restrictions:

1. We can only reason about closed terms
2. The equality is sometimes too intensional. For instance, lambdas are only equal if they are syntactically equal. This leaves no room for optimizations.

The notion of *contextual* (also *observational*) equivalence remedies both issues: Two programs M and N are contextually equivalent, if for any *closing context* (Fig. 2.5) $C[-]$ of natural number type, the terms $C[M]$ and $C[N]$ have the same operational behavior.

Here we only consider *termination to a natural number value* as observational behavior. This is not a restriction, we could have also chosen 1 or other ground types — the resulting notions of contextual equivalence are the same.

For this to give the correct notion of equality, we have to restrict our context to those, that preserve well-typedness. In Fig. 2.5 we give the typing rules for contexts.

We can now make precise what it means for programs to be contextual refinements.

Definition 2.2.13 (Contextual Refinement)**.** A term $M : \mathsf{Tm}^{\Gamma}_{\sigma}$ is a contextual refinement of $N : \mathsf{Tm}^{\Gamma}_{\sigma}$ if for any $C : (\Gamma \vdash \sigma) \Rightarrow (\cdot \vdash \mathsf{Nat})$ we have that

$$C[M] \Downarrow \underline{n} \to C[N] \Downarrow \underline{n}$$

We write $M \preceq_{\mathsf{Ctx}} N$ to denote that M contextually refines N.

$$\frac{\Gamma \vdash \sigma}{[-] : (\Gamma \vdash \sigma) \Rightarrow (\Gamma \vdash \sigma)} \qquad \frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mathsf{Nat})}{\mathsf{suc}(C) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mathsf{Nat})}$$

$$\frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mathsf{Nat})}{\mathsf{pred}(C) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mathsf{Nat})}$$

$$\frac{\Gamma' \vdash N : \tau \qquad \Gamma' \vdash M : \tau \qquad C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mathsf{Nat})}{\mathsf{ifz}(C,M,N) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}$$

$$\frac{\Gamma' \vdash L : \mathsf{Nat} \qquad \Gamma' \vdash N : \tau \qquad C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}{\mathsf{ifz}(L,C,N) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}$$

$$\frac{\Gamma' \vdash L : \mathsf{Nat} \qquad \Gamma' \vdash M : \tau \qquad C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}{\mathsf{ifz}(L,M,C) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}$$

$$\frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau) \qquad \Delta \vdash \tau'}{\mathsf{inl}(C) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau + \tau')} \qquad \frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau) \qquad \Delta \vdash \tau'}{\mathsf{inr}(C) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau' + \tau)}$$

$$\frac{\Delta, (x : \sigma_1) \vdash M : \tau \qquad \Delta, (y : \sigma_2) \vdash N : \tau \qquad C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \sigma_1 + \sigma_2)}{\mathsf{case}(C, x.M, y.N) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}$$

$$\frac{\Delta, (y : \sigma_2) \vdash N : \tau \qquad \Delta \vdash L : \sigma_1 + \sigma_2 \qquad C : (\Gamma \vdash \sigma) \Rightarrow (\Delta, (x : \sigma_1) \vdash \tau)}{\mathsf{case}(L, x.C, y.N) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}$$

$$\frac{\Delta, (x : \sigma_1) \vdash M : \tau \qquad \Delta \vdash L : \sigma_1 + \sigma_2 \qquad C : (\Gamma \vdash \sigma) \Rightarrow (\Delta, (y : \sigma_2) \vdash \tau)}{\mathsf{case}(L, x.M, y.C) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}$$

$$\frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau) \qquad \Delta \vdash M : \tau'}{\langle C, M \rangle : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau \times \tau')} \qquad \frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau') \qquad \Delta \vdash M : \tau}{\langle M, C \rangle : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau \times \tau')}$$

$$\frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau \times \tau')}{\mathsf{fst} C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)} \qquad \frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau \times \tau')}{\mathsf{snd} C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau')}$$

$$\frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta, x : \tau_1 \vdash \tau)}{\mathsf{lam}\ x.C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau_1 \to \tau)} \qquad \frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau \to \tau') \qquad \Delta \vdash M : \tau}{CM : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau')}$$

$$\frac{\Delta \vdash M : \tau \to \tau' \qquad C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)}{MC : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau')} \qquad \frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau[\mu X.\tau/X])}{\mathsf{fold} C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mu X.\tau)}$$

$$\frac{C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mu X.\tau)}{\mathsf{unfold} C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau[\mu X.\tau/X])}$$

Figure 2.5: Typed Context derivations

2.3 Denotational Semantics

The outline of this section is the following: We are first going to define semantic domains and an interpretation function by guarded recursion. As before, by using clock quantification we also get a collection of coinductive domains and interpretation functions therein. Then, we relate operational with denotational semantics. To do so, we prove three soundness theorems:

1. Theorem 2.3.6 relates the guarded evaluator to the guarded interpretation function.
2. Theorem 2.3.8 relates the coinductive evaluator to the coinductive interpretation function.
3. In Theorem 2.3.9 we prove that terminating programs evaluate to total elements in the coinductive domain (in the sense of the termination predicate).

We start by defining the semantic value domains $[\![-]\!]^\kappa : \mathsf{Ty}_\cdot \to \mathsf{U}$. To do so we use guarded recursion and induction on type derivations (Fig. 2.1).

Definition 2.3.1 (Semantic Values)**.** We define the following by guarded recursion and induction on type derivations.

$$[\![\sigma + \tau]\!]^\kappa \triangleq [\![\sigma]\!]^\kappa + [\![\tau]\!]^\kappa \qquad [\![\sigma \times \tau]\!]^\kappa \triangleq [\![\sigma]\!]^\kappa \times [\![\tau]\!]^\kappa$$
$$[\![\sigma \to \tau]\!]^\kappa \triangleq [\![\sigma]\!]^\kappa \to \mathsf{L}^\kappa([\![\tau]\!]^\kappa) \qquad [\![\mathsf{Nat}]\!]^\kappa \triangleq \mathbb{N}$$
$$[\![\mu X.\tau]\!]^\kappa \triangleq \triangleright(\alpha : \kappa).[\![\tau[\mu X.\tau/X]]\!]^\kappa \qquad [\![1]\!]^\kappa \triangleq 1$$

Using the partiality monad, semantic terms are then interpreted as lifted semantic values. Note, that by using guarded recursion, one can avoid defining an interpretation for open types.

The interpretation of open terms can only be meaningful if every variable is assigned a specific semantic value. This information is managed by *semantic environments*. What data is pushed into semantic environments is determined by the evaluation strategy — in our case CBV. Semantic environments are similar to substitutions and how they are realized is an implementation detail, closely tied to the implementation of variables and contexts.

Here, we represent them as partial functions from the set of all variables to the set of semantic values. We define semantic environments inductively by the following rules

$$\frac{}{\cdot : [\![\cdot]\!]^\kappa} \qquad \frac{\rho : [\![\Gamma]\!]^\kappa \qquad v : [\![\sigma]\!]^\kappa \qquad x \notin \Gamma}{\rho[x \mapsto v] : [\![\Gamma, x : \sigma]\!]^\kappa}$$

Instead of $\rho[x \mapsto v]$ we write $\rho.v$, if no ambiguity arises from that. Given $\rho : [\![\Gamma]\!]^\kappa$, we can look up the value for a variable $(x, \sigma) \in \Gamma$ in the following way

$$\boxed{\rho(x) : [\![\sigma]\!]^\kappa} \qquad \frac{}{\rho[x \mapsto v](x) = v} \qquad \frac{}{\rho[y \mapsto v](x) = \rho(x)}$$

Remark 19 (Permutation of environments). Just as for substitutions or contexts, handling permutations and weakenings of environments is another implementation detail, which we do not discuss in this paper. Thus, we simply note that looking up the entry for x in the environment ρ is invariant under permutation:

$$([x \mapsto v][y \mapsto w])(x) = ([y \mapsto w][x \mapsto v])(x).$$

Therefore, everything that follows inherits this invariance.

Interpretation function In Fig. 2.6 we define two interpretation functions

$$[\![-]\!]^{\mathsf{Val},\kappa}_{-} : \mathsf{Val}^{\Gamma}_{\sigma} \to [\![\Gamma]\!]^{\kappa} \to [\![\sigma]\!]^{\kappa}$$
$$[\![-]\!]^{\kappa}_{-} : \mathsf{Tm}^{\Gamma}_{\sigma} \to [\![\Gamma]\!]^{\kappa} \to \mathsf{L}^{\kappa}([\![\sigma]\!]^{\kappa})$$

which interpret values and terms respectively. The interpretation of terms is defined by induction on term derivations, but most of the time we write simply $[\![M]\!]^{\kappa}_{\rho}$ instead of $[\![\Gamma \vdash M : \sigma]\!]^{\kappa}_{\rho}$.

In the case of function application, we use the notation

$$d \cdot e \triangleq d >>=^{\kappa} \lambda f.e >>=^{\kappa} \lambda v.\mathsf{step}^{\kappa}(\lambda \alpha.fe)$$

Note that this introduces a step, similar to the coproduct elimination and unfolding of terms of the recursive type. For function application and coproduct elimination, the steps are not necessary. They are used to align the steps with the evaluator, which is needed to prove Theorem 2.3.6.

We proceed to define the value and term interpretation functions in Fig. 2.6.

Example 20. Recall the fixed point operator Y from Example 17. We can furthermore show that

$$[\![\mathsf{Y} f\, x]\!]^{\kappa}_{x \mapsto v} = (\mathsf{step}^{\kappa} \circ \mathsf{next}^{\kappa})^4([\![f(\mathsf{Y} f)\, x]\!]^{\kappa}_{x \mapsto v}) \tag{2.1}$$

for any value f and semantic value v.

Proof. Note that f is a value by assumption and Y as well as e_f are values by definition. Let $\rho \triangleq x \mapsto v$, and write Δ^{κ} for $\mathsf{step}^{\kappa} \circ \mathsf{next}^{\kappa}$.

$$\begin{aligned}
&[\![\mathsf{Y}(f)\, x]\!]^{\kappa}_{\rho} \\
&= ([\![\mathsf{Y}]\!]^{\kappa}_{\rho} \cdot [\![f]\!]^{\kappa}_{\rho}) \cdot [\![x]\!]^{\kappa}_{\rho} \\
&= \left((\Delta^{\kappa})[\![\mathsf{lam}\ z.e_f(\mathsf{fold}\,(e_f))z]\!]^{\kappa}_{\rho}\right) \cdot [\![x]\!]^{\kappa}_{\rho} \\
&= (\Delta^{\kappa})^2 \left([\![(e_f(\mathsf{fold}\,(e_f)))z]\!]^{\kappa}_{\rho, z \mapsto [\![x]\!]^{\mathsf{Val},\kappa}_{\rho}}\right) \\
&= (\Delta^{\kappa})^2 \left([\![e_f(\mathsf{fold}\,(e_f))]\!]^{\kappa}_{\rho} \cdot [\![x]\!]^{\kappa}_{\rho}\right) \\
&= (\Delta^{\kappa})^2 \left(\left([\![e_f]\!]^{\kappa}_{\rho} \cdot [\![\mathsf{fold}\,(e_f)]\!]^{\kappa}_{\rho}\right) \cdot [\![x]\!]^{\kappa}_{\rho}\right)
\end{aligned}$$

Interpretation of values

$$
\begin{aligned}
&[\![\underline{n}]\!]_\rho^{\mathsf{Val},\kappa} \triangleq n\\
&[\![\langle\rangle]\!]_\rho^{\mathsf{Val},\kappa} \triangleq \star\\
&[\![x]\!]_\rho^{\mathsf{Val},\kappa} \triangleq \rho(x)\\
&[\![\mathsf{lam}\ x.M]\!]_\rho^{\mathsf{Val},\kappa} \triangleq \lambda(v : [\![\sigma]\!]^\kappa).[\![\Gamma,(x:\sigma)\vdash M:\tau]\!]^\kappa_{\rho.v}\\
&[\![\langle V,W\rangle]\!]_\rho^{\mathsf{Val},\kappa} \triangleq ([\![V]\!]_\rho^{\mathsf{Val},\kappa},[\![W]\!]_\rho^{\mathsf{Val},\kappa})\\
&[\![\mathsf{inl}\,V]\!]_\rho^{\mathsf{Val},\kappa} \triangleq \mathbf{inl}\,[\![V]\!]_\rho^{\mathsf{Val},\kappa}\\
&[\![\mathsf{inr}\,V]\!]_\rho^{\mathsf{Val},\kappa} \triangleq \mathbf{inr}\,[\![V]\!]_\rho^{\mathsf{Val},\kappa}\\
&[\![\mathsf{fold}\,V]\!]_\rho^{\mathsf{Val},\kappa} \triangleq \mathsf{next}^\kappa[\![V]\!]_\rho^{\mathsf{Val},\kappa}
\end{aligned}
$$

Interpretation of terms

$$
\begin{aligned}
[\![\langle\rangle]\!]^\kappa_\rho &\triangleq \eta^\kappa(\star)\\
[\![\underline{n}]\!]^\kappa_\rho &\triangleq \eta^\kappa(n)\\
[\![\mathsf{suc}\,M]\!]^\kappa_\rho &\triangleq \mathsf{L}^\kappa(\mathbf{suc})\left([\![\Gamma\vdash M:\mathsf{Nat}]\!]^\kappa_\rho\right)\\
[\![\mathsf{pred}\,M]\!]^\kappa_\rho &\triangleq \mathsf{L}^\kappa(\mathbf{pred})\left([\![\Gamma\vdash M:\mathsf{Nat}]\!]^\kappa_\rho\right)\\
[\![\mathsf{ifz}(t,M,N)]\!]^\kappa_\rho &\triangleq [\![L]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \begin{cases} 0 & \mapsto [\![M]\!]^\kappa_\rho\\ n+1 & \mapsto [\![N]\!]^\kappa_\rho \end{cases}\\
[\![\mathsf{lam}\ x.M]\!]^\kappa_\rho &\triangleq \eta^\kappa\left(\lambda(v:[\![\sigma]\!]_\rho^{\mathsf{Val},\kappa}).[\![M]\!]^\kappa_{\rho.x\mapsto v}\right)\\
[\![MN]\!]^\kappa_\rho &\triangleq [\![M]\!]^\kappa_\rho\cdot[\![N]\!]^\kappa_\rho\\
[\![\langle M,N\rangle]\!]^\kappa_\rho &\triangleq [\![M]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \lambda v.\left([\![N]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \lambda w.(v,w)\right)\\
[\![\mathsf{fst}\,M]\!]^\kappa_\rho &\triangleq \mathsf{L}^\kappa(\mathsf{pr}_1)\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{snd}\,M]\!]^\kappa_\rho &\triangleq \mathsf{L}^\kappa(\mathsf{pr}_2)\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{inl}\,M]\!]^\kappa_\rho &\triangleq \mathsf{L}^\kappa(\mathbf{inl})\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{inr}\,M]\!]^\kappa_\rho &\triangleq \mathsf{L}^\kappa(\mathbf{inr})\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{case}(L,x.M,y.N)]\!]^\kappa_\rho &\triangleq [\![L]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \begin{cases} \mathbf{inl}\,v\mapsto \mathsf{step}^\kappa(\lambda\alpha.[\![M]\!]^\kappa_{\rho.x\mapsto v})\\ \mathbf{inr}\,v\mapsto \mathsf{step}^\kappa(\lambda\alpha.[\![N]\!]^\kappa_{\rho.y\mapsto v}) \end{cases}\\
[\![\mathsf{fold}\,M]\!]^\kappa_\rho &\triangleq \mathsf{L}^\kappa(\mathsf{next}^\kappa)([\![M]\!]^\kappa_\rho)\\
[\![\mathsf{unfold}\,M]\!]^\kappa_\rho &\triangleq [\![M]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda\alpha.\eta^\kappa(v[\alpha]))
\end{aligned}
$$

Figure 2.6: Interpretation of FPC.

$$
\begin{aligned}
&= (\Delta^\kappa)^3 \left(\begin{array}{r} [\![\text{let } y' = \mathsf{unfold}\,(\mathsf{fold}\, e_f) \text{ in } f(\mathsf{lam}\, x.(y'(\mathsf{fold}\, e_f))x)]\!]^\kappa_\rho \\ \cdot [\![x]\!]^\kappa_\rho \end{array} \right) \\
&= (\Delta^\kappa)^3 \left(\begin{array}{r} \left([\![\mathsf{lam}\, y'.f(\mathsf{lam}\, x.(y'(\mathsf{fold}\, e_f))x)]\!]^\kappa_\rho \cdot [\![\mathsf{unfold}\,(\mathsf{fold}\, e_f)]\!]^\kappa_\rho \right) \\ \cdot\ [\![x]\!]^\kappa_\rho \end{array} \right) \\
&= (\Delta^\kappa)^3 \left(\left((\Delta^\kappa)^2 [\![(f(\mathsf{lam}\, x.(y'(\mathsf{fold}\, e_f))x))]\!]^\kappa_{\rho, y' \mapsto [\![e_f]\!]^{\mathsf{Val},\kappa}_\rho} \right) \cdot [\![x]\!]^\kappa_\rho \right) \\
&= (\Delta^\kappa)^3 \left(\left((\Delta^\kappa)^2 \left([\![f]\!]^\kappa_\rho \cdot [\![\mathsf{lam}\, x.(e_f(\mathsf{fold}\, e_f))x]\!]^\kappa_\rho \right) \right) \cdot [\![x]\!]^\kappa_\rho \right) \\
&= (\Delta^\kappa)^3 \left(\left(\Delta^\kappa([\![f]\!]^\kappa_\rho \cdot [\![\mathsf{Y}(f)]\!]^\kappa_\rho) \right) \cdot [\![x]\!]^\kappa_\rho \right) \\
&= (\Delta^\kappa)^4 \left(\left([\![f(\mathsf{Y}(f))]\!]^\kappa_\rho \right) \cdot [\![x]\!]^\kappa_\rho \right) \\
&= (\Delta^\kappa)^4 \left([\![f(\mathsf{Y}(f))\, x]\!]^\kappa_\rho \right)
\end{aligned}
$$

Since $\mathsf{eval}^\kappa(\mathsf{Y} f V) = (\mathsf{step}^\kappa \circ \mathsf{next}^\kappa)^4(\mathsf{eval}^\kappa(f(\mathsf{Y} f) V))$ for any values f and V, we also get

$$[\![\mathsf{Y} f V]\!]^\kappa = (\mathsf{step}^\kappa \circ \mathsf{next}^\kappa)^4([\![f(\mathsf{Y} f)\, V]\!]^\kappa)$$

by Theorem 2.3.6 and Example 17. □

Lemma 2.3.2. *For any value $\Gamma \vdash V : \sigma$ and environment $\rho : [\![\Gamma]\!]^\kappa$ it is the case that*

$$[\![V]\!]^\kappa_\rho = \eta^\kappa([\![V]\!]^{\mathsf{Val},\kappa}_\rho)$$

Proof. We need to make a case analysis on all value constructors. All cases are immediate since both $>\!>\!=^\kappa$ and $\mathsf{L}^\kappa(f)$ reduce for values. □

In particular, for all values $\cdot \vdash V : \sigma \times \tau$ and $\cdot \vdash \underline{n} : \mathsf{Nat}$ we have

$$
\begin{aligned}
[\![\mathsf{fst}\, V]\!]^{\mathsf{Val},\kappa} &= \mathsf{pr}_1 [\![V]\!]^{\mathsf{Val},\kappa} & [\![\mathsf{snd}\, V]\!]^{\mathsf{Val},\kappa} &= \mathsf{pr}_2 [\![V]\!]^{\mathsf{Val},\kappa} \\
[\![\mathsf{suc}\ \underline{n}]\!]^{\mathsf{Val},\kappa} &= \mathbf{suc}[\![\underline{n}]\!]^{\mathsf{Val},\kappa} & [\![\mathsf{pred}\ \underline{n}]\!]^{\mathsf{Val},\kappa} &= \mathbf{pred}[\![\underline{n}]\!]^{\mathsf{Val},\kappa}
\end{aligned}
$$

Furthermore, the following properties are an immediate consequence of the uniqueness properties of coproduct elimination.

Lemma 2.3.3. *For any meta-level type A and terms $a, b : A$ we we have*

$$\begin{Bmatrix} \underline{0} & \mapsto a \\ \underline{n+1} & \mapsto b \end{Bmatrix} = \left(\begin{Bmatrix} 0 & \mapsto a \\ n+1 & \mapsto b \end{Bmatrix} \circ [\![-]\!]^{\mathsf{Val},\kappa} : \mathsf{Val}_{\mathsf{Nat}} \to A \right)$$

Lemma 2.3.4. *For functions $f : [\![\sigma]\!]^\kappa \to A$ and $g : [\![\tau]\!]^\kappa \to A$ we have*

$$\begin{Bmatrix} \mathsf{inl}\, V & \mapsto f([\![V]\!]^{\mathsf{Val},\kappa}) \\ \mathsf{inr}\, V & \mapsto g([\![V]\!]^{\mathsf{Val},\kappa}) \end{Bmatrix} = \left(\begin{Bmatrix} \mathbf{inl}\, v & \mapsto f \\ \mathbf{inr}\, v & \mapsto g \end{Bmatrix} \circ [\![-]\!]^{\mathsf{Val},\kappa} : \mathsf{Val}_{\sigma+\tau} \to A \right)$$

Substitution Lemma

Lemma 2.3.5 (Substitution Lemma)**.** *For any well-typed term* $\Gamma.(x:\sigma) \vdash M : \tau$ *as well as every well typed value* $\Gamma \vdash V : \sigma$ *we have*

$$[\![\Gamma \vdash M[V/x] : \tau]\!]^{\kappa}_{\rho} = [\![\Gamma.(x:\sigma) \vdash M : \tau]\!]^{\kappa}_{\rho.[\![\Gamma \vdash V:\sigma]\!]^{\mathsf{Val},\kappa}_{\rho}}$$

Proof. We proceed by induction on term derivations.

Case: $M = x$

Then

$$\begin{aligned} & [\![\Gamma \vdash x[V/x] : \tau]\!]^{\kappa}_{\rho} \\ = \; & [\![\Gamma \vdash V : \tau]\!]^{\kappa}_{\rho} \\ = \; & [\![\Gamma, (x:\sigma) \vdash x : \tau]\!]^{\kappa}_{\rho.[\![\Gamma \vdash V:\sigma]\!]^{\mathsf{Val},\kappa}_{\rho}} \end{aligned}$$

Case: $M = \mathsf{lam}\, x.M'$

Recall that we may permutate the environment as mentioned in Remark 19. Furthermore, assume without loss of generality that $y \neq x$. Otherwise, use an α-equivalent term where this condition is true (as discussed in Section 2.2). Consequently, we have that

$$\begin{aligned} & [\![\Gamma \vdash \mathsf{lam}\, x.M'[V/y] : \tau_1 \to \tau_2]\!]^{\kappa}_{\rho} \\ = \; & \eta^{\kappa}([\![\mathsf{lam}\, x.(M'[V/y])]\!]^{\mathsf{Val},\kappa}_{\rho}) \\ = \; & \eta^{\kappa}(\lambda(t : [\![\sigma]\!]^{\kappa}).[\![\Gamma, x:\tau_1 \vdash M'[V/y] : \tau_2]\!]^{\kappa}_{\rho,t}) \\ = \; & \eta^{\kappa}\left(\lambda(t : [\![\sigma]\!]^{\kappa}).[\![\Gamma, x:\tau_1, y:\sigma \vdash M' : \tau_2]\!]^{\kappa}_{\rho.[x\mapsto t].[y\mapsto [\![V]\!]^{\mathsf{Val},\kappa}_{\rho}]}\right) \\ = \; & \eta^{\kappa}\left(\lambda(t : [\![\sigma]\!]^{\kappa}).[\![\Gamma, y:\sigma, x:\tau_1 \vdash M' : \tau_2]\!]^{\kappa}_{\rho.[y\mapsto [\![V]\!]^{\mathsf{Val},\kappa}_{\rho}].[x\mapsto t]}\right) \\ = \; & \eta^{\kappa}\left([\![\Gamma, y:\sigma \vdash \mathsf{lam}\, x.M' : \tau_1 \to \tau_2]\!]^{\mathsf{Val},\kappa}_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_{\rho}}\right) \\ = \; & [\![\Gamma, y:\sigma \vdash \mathsf{lam}\, x.M' : \tau_1 \to \tau_2]\!]^{\kappa}_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_{\rho}} \end{aligned}$$

Case: $M = MN$

We have that

$$\begin{aligned} & [\![MN[V/x]]\!]^{\kappa}_{\rho} \\ = \; & [\![M[V/x]]\!]^{\kappa}_{\rho} \cdot [\![N[V/x]]\!]^{\kappa}_{\rho} \\ = \; & [\![M]\!]^{\kappa}_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_{\rho}} \cdot [\![N]\!]^{\kappa}_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_{\rho}} \\ = \; & [\![MN]\!]^{\kappa}_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_{\rho}} \end{aligned}$$

Case: $M = \mathsf{fold}\, M'$

We have that

$$[\![\Gamma \vdash \mathsf{fold}\, M'[V/x] : \mu X.\tau]\!]^{\kappa}_{\rho}$$

$$
\begin{aligned}
&= \mathsf{L}^\kappa(\mathsf{next}^\kappa)\left([\![\Gamma \vdash M'[V/x] : \tau[\mu X.\tau/X]]\!]^\kappa_\rho\right)\\
&= \mathsf{L}^\kappa(\mathsf{next}^\kappa)\left([\![\Gamma, x:\sigma \vdash M' : \tau[\mu X.\tau/X]]\!]^\kappa_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_\rho}\right)\\
&= [\![\Gamma, x:\sigma \vdash \mathsf{fold}\, M' : \mu X.\tau]\!]^\kappa_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_\rho}
\end{aligned}
$$

Case: $M = \mathsf{unfold}\, M'$
We have

$$
\begin{aligned}
&[\![\Gamma \vdash \mathsf{unfold}\, M'[V/x] : \tau[\mu X.\tau/X]]\!]^\kappa_\rho\\
&= [\![\Gamma \vdash M'[V/x] : \mu X.\tau]\!]^\kappa_\rho >>=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda\alpha.\eta^\kappa(v[\alpha]))\\
&= [\![\Gamma, x:\sigma \vdash M' : \mu X.\tau]\!]^\kappa_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_\rho} >>=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda\alpha.\eta^\kappa(v[\alpha]))\\
&= [\![\Gamma, x:\sigma \vdash \mathsf{unfold}\, M' : \tau[\mu X.\tau/X]]\!]^\kappa_{\rho.[\![V]\!]^{\mathsf{Val},\kappa}_\rho}
\end{aligned}
$$

We leave the remaining cases to the reader. □

Guarded soundness

The first of three soundness theorems relates the guarded evaluator with the guarded interpretation function.It relies on the fact that we synchronized the steps for these functions. The proof is technical, but many equalities are consequences of the monadic structure of L^κ and the fact that it is a delay algebra. These imply that $>>=^\kappa$, step^κ and $\mathsf{L}^\kappa(f)$ commute in many different ways.

Theorem 2.3.6 (Guarded soundness theorem)**.** *For any well typed closed expression* $\cdot \vdash M : \sigma$ *we have that*

$$\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa M) = [\![M]\!]^\kappa$$

Proof. First, assume the guarded hypothesis

$$\triangleright(\alpha:\kappa).\left(\forall M : \mathsf{Tm}_\sigma.\, \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa M) = [\![M]\!]^\kappa\right).$$

We proceed with case analysis of term derivations. **Case**: $M = V$ The value cases are all the same:

$$
\begin{aligned}
\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa V) &= \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\eta^\kappa(V))\\
&= \eta^\kappa([\![V]\!]^{\mathsf{Val},\kappa})\\
&= [\![V]\!]^\kappa
\end{aligned}
$$

Case: $M = \mathsf{suc}\, M'$

$$
\begin{aligned}
\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa(\mathsf{suc}\, M')) &= \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{L}^\kappa\mathbf{suc}(\mathsf{eval}^\kappa M')\right)\\
&= \mathsf{L}^\kappa([\![\mathbf{suc}(-)]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa M')
\end{aligned}
$$

$$
\begin{aligned}
&= \mathsf{L}^{\kappa}(\mathbf{suc}(\llbracket - \rrbracket^{\mathsf{Val},\kappa}))(\mathsf{eval}^{\kappa} M') \\
&= \mathsf{L}^{\kappa}(\mathbf{suc}) \left(\mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa} M') \right) \\
&= \mathsf{L}^{\kappa}(\mathbf{suc}) \left(\llbracket M' \rrbracket^{\kappa} \right) \\
&= \llbracket \mathsf{suc}\, M' \rrbracket^{\kappa}
\end{aligned}
$$

Case: $M = \mathsf{pred}\, M'$

$$
\begin{aligned}
\mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}(\mathsf{pred}\, M')) &= \mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa}) \left(\mathsf{L}^{\kappa}\mathbf{pred}(\mathsf{eval}^{\kappa} M') \right) \\
&= \mathsf{L}^{\kappa}(\llbracket \mathbf{pred}(-) \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa} M') \\
&= \mathsf{L}^{\kappa}(\mathbf{pred}(\llbracket - \rrbracket^{\mathsf{Val},\kappa}))(\mathsf{eval}^{\kappa} M') \\
&= \mathsf{L}^{\kappa}(\mathbf{pred}) \left(\mathsf{L}^{\kappa}\llbracket - \rrbracket^{\mathsf{Val},\kappa}(\mathsf{eval}^{\kappa} M') \right) \\
&= \mathsf{L}^{\kappa}(\mathbf{pred}) \left(\llbracket M' \rrbracket^{\kappa} \right) \\
&= \llbracket \mathsf{pred}\, M' \rrbracket^{\kappa}
\end{aligned}
$$

Case: $M = \mathsf{ifz}(L, M', N)$ Recall Lemma 2.3.3. It implies that

$$
\begin{cases}
\underline{0} \mapsto \mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}(M')) \\
\underline{n+1} \mapsto \mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}(N))
\end{cases}
= \lambda \underline{n}.\mathsf{match}\ \llbracket \underline{n} \rrbracket^{\mathsf{Val},\kappa}\ \mathsf{with} \begin{cases} 0 \mapsto \llbracket M' \rrbracket^{\kappa} \\ n+1 \mapsto \llbracket N \rrbracket^{\kappa} \end{cases}
$$

Consequently, we get that

$$
\begin{aligned}
&\mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}(\mathsf{ifz}(L, M', N))) \\
&= \mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa}) \left(\mathsf{eval}^{\kappa} L >>=^{\kappa} \begin{cases} \underline{0} \mapsto \mathsf{eval}^{\kappa}(M') \\ \underline{n+1} \mapsto \mathsf{eval}^{\kappa}(N) \end{cases} \right) \\
&= \left(\mathsf{eval}^{\kappa} L >>=^{\kappa} \begin{cases} \underline{0} \mapsto \mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}(M')) \\ \underline{n+1} \mapsto \mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}(N)) \end{cases} \right) \\
&= \mathsf{eval}^{\kappa} L >>=^{\kappa} \begin{cases} \underline{0} \mapsto \llbracket M' \rrbracket^{\kappa} \\ \underline{n+1} \mapsto \llbracket N \rrbracket^{\kappa} \end{cases} \\
&= \mathsf{eval}^{\kappa} L >>=^{\kappa} \lambda \underline{n}.\mathsf{match}\ \llbracket \underline{n} \rrbracket^{\mathsf{Val},\kappa}\ \mathsf{with} \begin{cases} 0 \mapsto \llbracket M' \rrbracket^{\kappa} \\ n+1 \mapsto \llbracket N \rrbracket^{\kappa} \end{cases} \\
&= \left(\mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa} L) \right) >>=^{\kappa} \begin{cases} 0 \mapsto \llbracket M' \rrbracket^{\kappa} \\ n+1 \mapsto \llbracket N \rrbracket^{\kappa} \end{cases} \\
&= \llbracket \mathsf{ifz}(L, M', N) \rrbracket^{\kappa}
\end{aligned}
$$

Case: $M = \langle M', N \rangle$

$$
\begin{aligned}
&\mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa} \langle M', N \rangle) \\
&= \mathsf{L}^{\kappa}(\llbracket - \rrbracket^{\mathsf{Val},\kappa}) \left(\mathsf{eval}^{\kappa} M' >>=^{\kappa} \lambda V.(\mathsf{eval}^{\kappa} N >>=^{\kappa} \lambda W.\eta^{\kappa}(\langle V, W \rangle)) \right)
\end{aligned}
$$

$$
\begin{aligned}
&= \mathsf{eval}^\kappa M' >>=^\kappa \lambda V.\left(\mathsf{eval}^\kappa N >>=^\kappa \lambda W.\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\,(\eta^\kappa(\langle V,W\rangle))\right)\\
&= \mathsf{eval}^\kappa M' >>=^\kappa \lambda V.\mathsf{eval}^\kappa N >>=^\kappa \lambda W.\eta^\kappa([\![\langle V,W\rangle]\!]^{\mathsf{Val},\kappa})\\
&= \mathsf{eval}^\kappa M' >>=^\kappa \lambda V.\mathsf{eval}^\kappa N >>=^\kappa \lambda W.\eta^\kappa(([\![V]\!]^{\mathsf{Val},\kappa},[\![W]\!]^{\mathsf{Val},\kappa}))\\
&= \begin{array}{l}(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa M')) >>=^\kappa \lambda v.(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa N))\\ \qquad\qquad >>=^\kappa \lambda w.\eta^\kappa((v,w))\end{array}\\
&= [\![M']\!]^\kappa >>=^\kappa \lambda v.[\![N]\!]^\kappa >>=^\kappa \lambda w.\eta^\kappa((v,w))\\
&= [\![\langle M',N\rangle]\!]^\kappa
\end{aligned}
$$

Case: $M = \mathsf{fst}\,M'$

$$
\begin{aligned}
&\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa(\mathsf{fst}\,M'))\\
&= \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{L}^\kappa(\underline{\pi_1})(\mathsf{eval}^\kappa M')\right)\\
&= \mathsf{L}^\kappa([\![\underline{\pi_1}(-)]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa M'\right)\\
&= \mathsf{L}^\kappa(\mathsf{pr}_1)\left(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa M'\right)\right)\\
&= \mathsf{L}^\kappa(\mathsf{pr}_1)\left([\![M']\!]^\kappa\right)\\
&= [\![\mathsf{fst}\,M']\!]^\kappa
\end{aligned}
$$

Case: $M = \mathsf{snd}\,M'$

$$
\begin{aligned}
&\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa(\mathsf{snd}\,M'))\\
&= \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{L}^\kappa(\underline{\pi_2})(\mathsf{eval}^\kappa M')\right)\\
&= \mathsf{L}^\kappa([\![\underline{\pi_2}(-)]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa M'\right)\\
&= \mathsf{L}^\kappa(\mathsf{pr}_2)\left(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa M'\right)\right)\\
&= \mathsf{L}^\kappa(\mathsf{pr}_2)\left([\![M']\!]^\kappa\right)\\
&= [\![\mathsf{snd}\,M']\!]^\kappa
\end{aligned}
$$

Case: $M = MN$ We write Δ^κ for $\mathsf{step}^\kappa \circ \mathsf{next}^\kappa$. We start by unfolding the definition of $(\mathsf{eval}^\kappa MN) >>=^\kappa \eta^\kappa \circ [\![-]\!]^{\mathsf{Val},\kappa}$ and use associativity as well as the definition of $>>=^\kappa$ to get

$$
\begin{aligned}
&\left(\mathsf{eval}^\kappa M >>=^\kappa (\lambda(\mathsf{lam}\,x.M').\mathsf{eval}^\kappa N >>=^\kappa \right.\\
&\qquad \left.\lambda V.(\Delta^\kappa(\mathsf{eval}^\kappa(M'[V/x]))))\right) >>=^\kappa \eta^\kappa \circ [\![-]\!]^{\mathsf{Val},\kappa}\\
&= \left(\begin{array}{l}\mathsf{eval}^\kappa M >>=^\kappa \lambda(\mathsf{lam}\,x.M').\mathsf{eval}^\kappa N\\ \qquad >>=^\kappa \lambda V.(\Delta^\kappa\left(\mathsf{eval}^\kappa(M'[V/x])\right) >>=^\kappa \eta^\kappa \circ [\![-]\!]^{\mathsf{Val},\kappa})\end{array}\right)\\
&= \left(\begin{array}{l}\mathsf{eval}^\kappa M >>=^\kappa \lambda(\mathsf{lam}\,x.M').\mathsf{eval}^\kappa N\\ \qquad >>=^\kappa \lambda V.\Delta^\kappa\left(\mathsf{eval}^\kappa(M'[V/x]) >>=^\kappa \eta^\kappa \circ [\![-]\!]^{\mathsf{Val},\kappa}\right)\end{array}\right)\\
&= \left(\begin{array}{l}\mathsf{eval}^\kappa M >>=^\kappa \lambda(\mathsf{lam}\,x.M').\mathsf{eval}^\kappa N\\ \qquad >>=^\kappa \lambda V.(\mathsf{step}^\kappa(\lambda\alpha.[\![M'[V/x]]\!]^\kappa_\rho))\end{array}\right)
\end{aligned}
$$

In the last step, we applied the guarded hypothesis. Next, we use the substitution lemma for terms, which leaves us with

$$\begin{aligned}
&= \mathsf{eval}^\kappa M >>=^\kappa \left(\lambda(\mathsf{lam}\ x.M').\mathsf{eval}^\kappa N >>=^\kappa \lambda W.(\Delta^\kappa([\![\ M']\!]^\kappa_{[\![W]\!]^{\mathsf{Val},\kappa}}))\right)\\
&= \begin{pmatrix}\mathsf{eval}^\kappa M >>=^\kappa \big(\lambda(\mathsf{lam}\ x.M').\mathsf{eval}^\kappa N\\ \qquad >>=^\kappa \lambda W.(\Delta^\kappa([\![\mathsf{lam}\ x.M']\!]^{\mathsf{Val},\kappa}([\![W]\!]^{\mathsf{Val},\kappa})))\big)\end{pmatrix}\\
&= \mathsf{eval}^\kappa M >>=^\kappa \left(\lambda V.\mathsf{eval}^\kappa N >>=^\kappa \lambda W.(\Delta^\kappa([\![V]\!]^{\mathsf{Val},\kappa}([\![W]\!]^{\mathsf{Val},\kappa})))\right)
\end{aligned}$$

In the last equation, we simply omitted the case analysis of function values (as we do not refer to the body of the function). We now use the functoriality of L^κ to get

$$\begin{aligned}
= (\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa M)) >>=^\kappa \big(&\lambda v.(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa N))\\
&>>=^\kappa \lambda w.\Delta^\kappa(vw)\big)
\end{aligned}$$

which by the induction hypothesis gives

$$\begin{aligned}
&= [\![M]\!]^\kappa >>=^\kappa \left(\lambda v.[\![N]\!]^\kappa_\cdot >>=^\kappa \lambda w.\mathsf{step}^\kappa(\lambda(\alpha:\kappa).vw)\right)\\
&= [\![M]\!]^\kappa \cdot [\![N]\!]^\kappa\\
&= [\![MN]\!]^\kappa
\end{aligned}$$

Case: $M = \mathsf{inl}\, M'$

$$\begin{aligned}
&\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa(\mathsf{inl}\, M')\right)\\
&= \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{L}^\kappa(\mathbf{inl})\left(\mathsf{eval}^\kappa M'\right)\right)\\
&= \mathsf{L}^\kappa([\![\mathsf{inl}\,(-)]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa M'\right)\\
&= \mathsf{L}^\kappa(\mathbf{inl})\left(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa M'\right)\right)\\
&= \mathsf{L}^\kappa(\mathbf{inl})\left([\![M']\!]^\kappa\right)\\
&= [\![\mathsf{inl}\, M']\!]^\kappa
\end{aligned}$$

Case: $M = \mathsf{inr}\, M'$

The case is completely analogous to the previous case.

Case: $M = \mathsf{case}(L, x.M, y.N)$

We proceed similarly to the function application case and first unfold the definition of $\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\,(\mathsf{eval}^\kappa \mathsf{case}(L, x.M, y.N))$

$$\begin{aligned}
&\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa L >>=^\kappa \begin{cases}\mathsf{inl}\, V \mapsto (\Delta^\kappa \mathsf{eval}^\kappa(M[V/x]))\\ \mathsf{inr}\, V \mapsto (\Delta^\kappa \mathsf{eval}^\kappa(N[V/x]))\end{cases}\right)\\
&= \mathsf{eval}^\kappa L >>=^\kappa \begin{cases}\mathsf{inl}\, V \mapsto \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\,(\Delta^\kappa(\mathsf{eval}^\kappa(M[V/x])))\\ \mathsf{inr}\, V \mapsto \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\,(\Delta^\kappa(\mathsf{eval}^\kappa(N[V/x])))\end{cases}\\
&= \mathsf{eval}^\kappa L >>=^\kappa \begin{cases}\mathsf{inl}\, V \mapsto \Delta^\kappa\big(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\,(\mathsf{eval}^\kappa(M[V/x]))\big)\\ \mathsf{inr}\, V \mapsto \Delta^\kappa\big(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\,(\mathsf{eval}^\kappa(N[V/x]))\big)\end{cases}
\end{aligned}$$

$$= \mathsf{eval}^\kappa L >>=^\kappa \begin{cases} \mathsf{inl}\, V \mapsto (\Delta^\kappa [\![M[V/x]]\!]^\kappa) \\ \mathsf{inr}\, V \mapsto (\Delta^\kappa [\![N[V/x]]\!]^\kappa) \end{cases}$$

We now apply the substitution lemma and use Lemma 2.3.4. Finally, we apply the functoriality of $\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})$ and get the result.

$$\begin{aligned}
&= \mathsf{eval}^\kappa L >>=^\kappa \begin{cases} \mathsf{inl}\, V \mapsto \Delta^\kappa([\![M]\!]^\kappa_{[\![V]\!]^{\mathsf{Val},\kappa}}) \\ \mathsf{inr}\, V \mapsto \Delta^\kappa([\![N]\!]^\kappa_{[\![V]\!]^{\mathsf{Val},\kappa}}) \end{cases} \\
&= \mathsf{eval}^\kappa L >>=^\kappa \lambda V.\mathsf{match}\ [\![V]\!]^{\mathsf{Val},\kappa}\ \mathsf{with} \begin{cases} \mathbf{inl}\, v \mapsto (\Delta^\kappa [\![M]\!]^\kappa_v) \\ \mathbf{inr}\, v \mapsto (\Delta^\kappa [\![N]\!]^\kappa_v) \end{cases} \\
&= \left(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^\kappa L)\right) >>=^\kappa \begin{cases} \mathbf{inl}\, v \mapsto \Delta^\kappa([\![M]\!]^\kappa_v) \\ \mathbf{inr}\, v \mapsto \Delta^\kappa([\![N]\!]^\kappa_v) \end{cases} \\
&= [\![L]\!]^\kappa >>=^\kappa \begin{cases} \mathbf{inl}\, v \mapsto \Delta^\kappa([\![M]\!]^\kappa_v) \\ \mathbf{inr}\, v \mapsto \Delta^\kappa([\![N]\!]^\kappa_v) \end{cases} \\
&= [\![\mathsf{case}(L, x.M, y.N)]\!]^\kappa
\end{aligned}$$

Case: $M = \mathsf{fold}\, M'$

$$\begin{aligned}
&\left(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\right)\left(\mathsf{eval}^\kappa(\mathsf{fold}\, M')\right) \\
&= \left(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{L}^\kappa(\mathsf{fold})\right)\left(\mathsf{eval}^\kappa(M')\right)\right) \\
&= \left(\mathsf{L}^\kappa([\![\mathsf{fold}\, -]\!]^{\mathsf{Val},\kappa})\right)\left(\mathsf{eval}^\kappa(M')\right) \\
&= \left(\mathsf{L}^\kappa(\mathsf{next}^\kappa \circ [\![-]\!]^{\mathsf{Val},\kappa})\right)\left(\mathsf{eval}^\kappa(M')\right) \\
&= \left(\mathsf{L}^\kappa(\mathsf{next}^\kappa)\right)\left(\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa(M')\right)\right) \\
&= \left(\mathsf{L}^\kappa(\mathsf{next}^\kappa)\right)\left([\![M']\!]^\kappa\right) \\
&= [\![\mathsf{fold}\, M']\!]^\kappa
\end{aligned}$$

Case: $M = \mathsf{unfold}\, M'$
Note that for $V' : \mathsf{Val}_{\tau[\mu \mathsf{X}.\tau/\mathsf{X}]}$ we have that

$$\triangleright(\alpha : \kappa).\left((([\![\mathsf{fold}\, V']\!]^{\mathsf{Val},\kappa})\,[\alpha] = [\![V']\!]^{\mathsf{Val},\kappa}\right)$$

and thus we get

$$\begin{aligned}
&\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa(\mathsf{unfold}\, M')\right) \\
&= \mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa(M') >>=^\kappa \lambda(\mathsf{fold}\, V').\Delta^\kappa(\eta^\kappa(V'))\right) \\
&= \mathsf{eval}^\kappa(M') >>=^\kappa \lambda(\mathsf{fold}\, V').\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\Delta^\kappa(\eta^\kappa(V'))) \\
&= \mathsf{eval}^\kappa(M') >>=^\kappa \lambda(\mathsf{fold}\, V').\mathsf{L}^\kappa([\![-]\!]^{\mathsf{Val},\kappa})(\Delta^\kappa(\eta^\kappa([\![V']\!]^{\mathsf{Val},\kappa})))
\end{aligned}$$

$$= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\, V').(\mathsf{step}^\kappa(\lambda(\alpha:\kappa).(\eta^\kappa([\![V']\!]^{\mathsf{Val},\kappa})))$$
$$= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\, V').(\mathsf{step}^\kappa(\lambda(\alpha:\kappa).(\eta^\kappa([\![\mathsf{fold}\, V']\!]^{\mathsf{Val},\kappa}\,[\alpha]))))$$

The case analysis of the constructor $\mathsf{fold}\, V'$ is now superfluous, and thus we get

$$= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda V.(\mathsf{step}^\kappa(\lambda(\alpha:\kappa).(\eta^\kappa([\![V]\!]^{\mathsf{Val},\kappa}\,[\alpha]))))$$
$$= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa (\lambda v.\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\eta^\kappa(v\,[\alpha]))) \circ [\![-]\!]^{\mathsf{Val},\kappa}$$
$$= (\mathsf{L}^\kappa[\![-]\!]^{\mathsf{Val},\kappa}(\mathsf{eval}^\kappa(M'))) >\!\!>\!\!=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\eta^\kappa(v\,[\alpha]))$$
$$= [\![\mathsf{unfold}\, M']\!]^\kappa$$

□

Coinductive Semantics

As a first observation in this section, we note that $[\![-]\!]^\kappa$ and $\mathsf{L}^\kappa[\![-]\!]^\kappa$ are the guarded recursive fixpoints of the mutually defined Ty-indexed endofunctors F and G.

$$\begin{aligned} F(P,\mathsf{Nat}) &\triangleq \mathbb{N} \\ F(P,\sigma\times\tau) &\triangleq F(P,\sigma)\times F(P,\tau) \\ &\vdots \qquad\qquad\qquad\qquad G(Q,\sigma) \triangleq F(P,\sigma)+Q(\sigma) \\ F(P,\sigma\to\tau) &\triangleq F(P,\sigma)\to G(Q,\tau) \\ F(P,\mu X.\tau) &\triangleq P(\tau[\mu X.\tau/X]) \end{aligned}$$

Lemma 2.3.7. *We have that* $\nu^\kappa(F) = [\![-]\!]^\kappa$ *and* $\nu^\kappa(G) = \mathsf{L}^\kappa([\![-]\!]^\kappa)$.

Proof. We proceed by guarded recursion, to that goal assume that $\triangleright^\kappa(\nu^\kappa(F) = [\![-]\!]^\kappa)$ as well as $\triangleright^\kappa(\nu^\kappa(G) = \mathsf{L}^\kappa([\![-]\!]^\kappa))$.

It suffices now to prove that $\nu^\kappa(F)(\sigma) = [\![\sigma]\!]^\kappa$ for all $\sigma : \mathsf{Ty}$, and thus we proceed by induction on σ.

If $\sigma = \sigma_1 \to \tau$, then

$$\begin{aligned} \nu^\kappa(F)(\sigma) &= F(\triangleright^\kappa(\nu^\kappa(F)),\sigma_1) \to G(\triangleright^\kappa(\nu^\kappa(G)),\tau) \\ &= \nu^\kappa(F)(\sigma_1) \to (F(\triangleright^\kappa(\nu^\kappa(F)),\tau) + \triangleright^\kappa(\nu^\kappa(G))(\tau)) \\ &= \nu^\kappa(F)(\sigma_1) \to (\nu^\kappa(F)(\tau) + \triangleright^\kappa(\nu^\kappa(G))(\tau)) \\ &= [\![\sigma_1]\!]^\kappa \to ([\![\tau]\!]^\kappa + \triangleright^\kappa(\mathsf{L}^\kappa[\![\tau]\!]^\kappa)) \quad \text{by guarded and inductive hypothesis} \\ &= [\![\sigma_1]\!]^\kappa \to \mathsf{L}^\kappa[\![\tau]\!]^\kappa \\ &= [\![\sigma]\!]^\kappa \end{aligned}$$

If $\sigma = \mu X.\tau$, then

$$\begin{aligned} \nu^\kappa(F)(\sigma) &= F(\triangleright^\kappa(\nu^\kappa(F)),\mu X.\tau) \\ &= \triangleright^\kappa(\nu^\kappa(F))(\tau[\mu X.\tau/X]) \end{aligned}$$

$$
\begin{aligned}
&= \triangleright^{\kappa}([\![\tau[\mu X.\tau/X]]\!]^{\kappa}) \\
&= [\![\mu X.\tau]\!]^{\kappa}
\end{aligned}
$$

The remaining cases and the fact that $\nu^{\kappa}(G) = \mathsf{L}^{\kappa}([\![-]\!]^{\kappa})$ follow similarly. □

Following Theorem 1.3.17, F and G commute with clock quantification and therefore Theorem 1.3.14 implies that they have a coinductive solution as well. We denote with $[\![-]\!] \triangleq \forall\kappa.[\![-]\!]^{\kappa}$ and $\forall\kappa.\mathsf{L}^{\kappa}([\![-]\!]^{\kappa})$ the coinductive solutions for F and G respectively.

Note that if $[\![\sigma]\!]^{\kappa}$ is clock irrelevant, then we get that $\forall\kappa.\mathsf{L}^{\kappa}([\![\sigma]\!]^{\kappa}) \simeq \mathsf{L}([\![\sigma]\!])$. In particular, we have that $\forall\kappa.\mathsf{L}^{\kappa}([\![\mathsf{Nat}]\!]^{\kappa}) \simeq \mathsf{L}([\![\mathsf{Nat}]\!])$.

Since clock quantification is an applicative functor and both $\mathsf{Val}^{\Gamma}_{\sigma}$ and $\mathsf{Tm}^{\Gamma}_{\sigma}$ are clock-irrelevant, we inherit the interpretation functions $[\![-]\!]^{\mathsf{Val}}_{-} : \mathsf{Val}^{\Gamma}_{\sigma} \to \forall\kappa.[\![\Gamma]\!]^{\kappa} \to [\![\sigma]\!]$ as well as $[\![-]\!]_{-} : \mathsf{Tm}^{\Gamma}_{\sigma} \to \forall\kappa.[\![\Gamma]\!]^{\kappa} \to \forall\kappa.\mathsf{L}^{\kappa}([\![\sigma]\!]^{\kappa})$ by letting:

$$
\begin{aligned}
[\![V]\!]^{\mathsf{Val}}_{\rho} &\triangleq \Lambda\kappa.[\![V]\!]^{\mathsf{Val},\kappa}_{\rho[\kappa]} \\
[\![M]\!]_{\rho} &\triangleq \Lambda\kappa.[\![M]\!]^{\kappa}_{\rho[\kappa]}
\end{aligned}
$$

Now, it is easy to prove the following soundness theorem.

Theorem 2.3.8 (Coinductive Soundness Theorem)**.** *For $\cdot \vdash M : \sigma$ we have that*

$$
\left(\Lambda\kappa.\mathsf{L}^{\kappa}([\![-]\!]^{\mathsf{Val},\kappa})\right)(\mathsf{eval}(M)) = [\![M]\!]
$$

Proof. It follows from Theorem 2.3.6 that $\mathsf{L}^{\kappa}([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}M) = [\![M]\!]^{\kappa}$ and consequently also $\forall\kappa.\left(\mathsf{L}^{\kappa}([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}M) = [\![M]\!]^{\kappa}\right)$. This is however equivalent to $\Lambda\kappa.(\mathsf{L}^{\kappa}([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}M)) = [\![M]\!]$ by Fig. 1.7.

Finally, since $\forall\kappa$ is an applicative functor,

$$
\Lambda\kappa.(\mathsf{L}^{\kappa}([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}M)) = \left(\Lambda\kappa.\mathsf{L}^{\kappa}([\![-]\!]^{\mathsf{Val},\kappa})\right)(\mathsf{eval}(M))
$$

and the claim follows. □

Now we prove that FPC-programs $\vdash M : \mathsf{Nat}$, which terminate to a value $\underline{n}$, are interpreted by total denotations, meaning that $[\![M]\!] \Downarrow^{\forall} n$.

Theorem 2.3.9. *For any well typed closed term $\cdot \vdash M : \mathsf{Nat}$ we have that*

$$
M \Downarrow \underline{n} \to [\![M]\!] \Downarrow^{\forall} n
$$

Proof. Recall that $[\![\underline{n}]\!]^{\mathsf{Val}} = n$

By Lemma 2.2.12 we have that $M \Downarrow \underline{n} \to \mathsf{eval}(M) \Downarrow^{\forall} \underline{n}$. Furthermore, $\eta^{\forall} \circ [\![-]\!]^{\mathsf{Val}} : \mathsf{Val}_{\mathsf{Nat}} \to \mathsf{L}([\![\mathsf{Nat}]\!])$ and $\eta^{\forall}([\![\underline{n}]\!]^{\mathsf{Val}}) \Downarrow^{\forall} n$. Thus, Corollary 2.2.11 implies that $\mathsf{L}([\![-]\!]^{\mathsf{Val}})(\mathsf{eval}(M)) \Downarrow^{\forall} n$. Now Theorem 2.3.8 gives that $[\![M]\!] \Downarrow^{\forall} n$ □

2.4 Lifting Relations

A relation $\mathscr{R} : A \to \mathsf{Val}_\sigma \to \mathsf{Prop}$ can be lifted to a relation $\overline{\mathscr{R}}^\kappa : \mathsf{L}^\kappa(A) \to \mathsf{Tm}_\sigma \to \mathsf{Prop}$ (note that nothing prevents $\mathscr{R}$ and A to also depend on κ). This is done in two steps

1. First, we lift a relation $\mathscr{R} : A \to \mathsf{Val}_\sigma \to \mathsf{Prop}$ to a relation $\mathscr{R}^{\mathsf{tm}} : A \to \mathsf{Tm}_\sigma \to \mathsf{Prop}$ by letting $a \mathscr{R}^{\mathsf{tm}} M \triangleq \exists V. M \Downarrow V \wedge a \mathscr{R} V$

2. Then, we extend $\mathscr{R}^{\mathsf{Tm}}$ to $\mathsf{L}^\kappa(A)$ by letting

$$d \overline{\mathscr{R}}^\kappa M \triangleq \left(d >>=^\kappa \lambda v. v \mathscr{R}^{\mathsf{Tm}} M \right)$$

In the following, we show that the relational lifting respects the monad structure of L^κ.

Lemma 2.4.1. *For all $a : A$ and $V : \mathsf{Val}_\sigma$ we have $a \mathscr{R} V \simeq (\eta^\kappa(a)) \overline{\mathscr{R}}^\kappa V$.*

Proof. The proof is immediate by the definition of $>>=^\kappa$ and the fact that $V \Downarrow V$ if V is a value. □

Lemma 2.4.2. *For any $d : \triangleright^\kappa \mathsf{L}^\kappa(A)$ and $V : \mathsf{Val}_\sigma$ it is the case that*

$$(\mathsf{step}^\kappa(\lambda(\alpha : \kappa).d)) \overline{\mathscr{R}}^\kappa V \simeq \triangleright(\alpha : \kappa).(d[\alpha] \overline{\mathscr{R}}^\kappa V)$$

Proof. The lemma follows from the fact that for any V the function $- \overline{\mathscr{R}}^\kappa V : \mathsf{L}^\kappa(A) \to \mathsf{Prop}$ is a delay algebra homomorphism. □

Lemma 2.4.3. *Let $M \rightsquigarrow^* M'$, then for any $d : \mathsf{L}^\kappa(A)$ we have*

$$d \overline{\mathscr{R}}^\kappa M \simeq d \overline{\mathscr{R}}^\kappa M'$$

Proof. The proof proceeds by guarded recursion. Both implications are similar, so we only prove one direction. Assume $d \overline{\mathscr{R}}^\kappa M$.

Case: $d = \eta^\kappa a$
Then $\eta^\kappa(a) \overline{\mathscr{R}}^\kappa M$ directly reduces to $\exists V. M \Downarrow V \wedge a \mathscr{R} V$. Now Corollary 2.2.8 implies that $M' \Downarrow V$ and it follows that $\exists V. M \Downarrow V \wedge a \mathscr{R} V$.

Case: $d = \mathsf{step}^\kappa d'$
Lemma 2.4.2 implies that $\mathsf{step}^\kappa d' \overline{\mathscr{R}}^\kappa M$ is equivalent to $\triangleright(\alpha : \kappa).d'[\alpha] \overline{\mathscr{R}}^\kappa M$, and thus we can apply the guarded hypothesis to get the result. □

Lemma 2.4.4. *Let E denote an evaluation context (as specified by the grammar in Section 2.2). If $f : A \to \mathsf{L}^\kappa(B)$ and $E : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$ satisfy $f(a) \overline{\mathscr{S}}^\kappa E[V]$ whenever $a \mathscr{R} V$, then for all $d : \mathsf{L}^\kappa(A)$ and $M : \mathsf{Tm}_\sigma$ satisfying $d \overline{\mathscr{R}}^\kappa M$, also $\overline{f}(d) \overline{\mathscr{S}}^\kappa E[M]$.*

Proof. We proceed by guarded recursion.

Case: $d = \eta^\kappa(a)$

In this case, $\eta^\kappa(a)\,\overline{\mathscr{R}}^\kappa\,M$ reduces to $\exists V.M \Downarrow V \wedge a\mathscr{R}V$. This implies that $f(a)\,\overline{\mathscr{S}}^\kappa\,E[V]$. By Lemma 2.2.9 we have that $M \rightsquigarrow^* V$ and thus also $E[M] \rightsquigarrow^* E[V]$. Now Lemma 2.4.3 gives us $f(a)\,\overline{\mathscr{S}}^\kappa\,E[M]$.

Case: $d = \mathsf{step}^\kappa d'$

Lemma 2.4.2 implies that $\mathsf{step}^\kappa(d')\,\overline{\mathscr{R}}^\kappa\,M$ is equivalent to $\triangleright(\alpha:\kappa).d'[\alpha]\,\overline{\mathscr{R}}^\kappa\,M$. Thus, we get that $\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\overline{f}(d'[\alpha]))\,\overline{\mathscr{S}}^\kappa\,E[M]$ by applying the guarded hypothesis. Since $\overline{f}$ is a delay algebra homomorphism, it commutes with step^κ and thus the result follows. □

Corollary 2.4.5. *Let E denote an evaluation context (as specified by the grammar in Section 2.2). If $f : A \to B$ and $E : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$ satisfy $\exists W.E[V] \Downarrow W \wedge f(a)\ \mathscr{S}\ W$ whenever $a\mathscr{R}V$, then for all $d : \mathsf{L}^\kappa(A)$ and $M : \mathsf{Tm}_\sigma$ satisfying $d\,\overline{\mathscr{R}}^\kappa\,M$, also $(\mathsf{L}^\kappa(f))(d)\,\overline{\mathscr{S}}^\kappa\,E[M]$.*

Proof. We apply Lemma 2.4.4 to the function $f' \triangleq \eta^\kappa \circ f$ and evaluation context E. Note that $\eta^\kappa(f(a))\,\overline{\mathscr{S}}^\kappa\,E[V]$ is equivalent to $\exists W.E[V] \Downarrow W \wedge f(a)\ \mathscr{S}\ W$. □

Coinductive Relation Lifting For a relation $\mathscr{R} : A \to \mathsf{Val}_\sigma \to \mathsf{Prop}$ we define $-\,\overline{\mathscr{R}}\,- : \forall\kappa.\mathsf{L}^\kappa(A) \to \mathsf{Tm}_\sigma \to \mathsf{Prop}$ by

$$d\ \overline{\mathscr{R}}\ M \triangleq \forall\kappa.(d[\kappa]\,\overline{\mathscr{R}}^\kappa\,M).$$

If A and $\mathscr{R}$ are clock irrelevant, then $\overline{\mathscr{R}}$ is the greatest fixpoint of the $\mathsf{L}(A) \times \mathsf{Tm}_\sigma$-indexed endofunctor

$$F_{\mathscr{R}} : ((\mathsf{L}(A) \times \mathsf{Tm}_\sigma) \to \mathsf{Prop}) \to (\mathsf{L}(A) \times \mathsf{Tm}_\sigma) \to \mathsf{Prop}$$
$$F_{\mathscr{R}}(P)(\eta^\forall a, M) \triangleq \forall\kappa.\left(a\mathscr{R}^{\mathsf{tm}}M\right) \qquad F_{\mathscr{R}}(P)(\mathsf{step}^\forall d, M) \triangleq P(d,M),$$

To see this, we first prove that $\overline{\mathscr{R}}^\kappa$ is the guarded fixpoint $\nu^\kappa(F_{\mathscr{R}})$.

Lemma 2.4.6. *Let $\mathscr{S}^\kappa \triangleq \nu^\kappa(F_{\mathscr{R}})$, then for all $d : \mathsf{L}(A)$ we have that*

$$d[\kappa]\,\overline{\mathscr{R}}^\kappa\,M \simeq d\ \mathscr{S}^\kappa\ M$$

Proof. We proceed by guarded recursion, to that goal, assume that

$$\triangleright^\kappa\left(\forall d : \mathsf{L}(A).\left(d[\kappa]\,\overline{\mathscr{R}}^\kappa\,M \simeq d\ \mathscr{S}^\kappa\ M\right)\right)$$

Now, we case on $d : \mathsf{L}(A)$, which gives the following.

Case: $d = \eta^\forall(a)$

This case is immediate since both sides reduce to $a\mathscr{R}^{\mathsf{Tm}}M$.

Case: $d = \mathsf{step}^\forall(d')$

For this case, we have to prove that $\triangleright^\kappa(d'[\kappa]\,\overline{\mathscr{R}}^\kappa\,M) \simeq \triangleright^\kappa(d'\ \mathscr{S}^\kappa\ M)$ But this follows immediately from the guarded hypothesis. □

Lemma 2.4.7. *$\overline{\mathscr{R}}$ is the coinductive solution of $F_{\mathscr{R}}$ (Lemma 2.4.6). As such it is the case that:*

- $\eta^{\forall} a\ \overline{\mathscr{R}}\ M \simeq a \mathscr{R}^{\mathsf{Tm}} M$.
- $\mathsf{step}^{\forall}(d)\ \overline{\mathscr{R}}\ M \simeq d\ \overline{\mathscr{R}}\ M$

Proof. Theorem 1.3.17 implies that $F_{\mathscr{R}}$ is clock-irrelevant. Then, the claim follows from Lemma 2.4.6. □

As the first application of this lifting, we show that the lifted equality relation can be used to prove that a natural number program M terminates with value $\underline{n}$.

Lemma 2.4.8. *Let $-\mathsf{eq}_{\mathsf{Nat}}- : \mathbb{N} \times \mathsf{Val}_{\mathsf{Nat}} \to \mathsf{Prop}$ denote equality modulo the canonical isomorphism $\mathsf{Val}_{\mathsf{Nat}} \simeq \mathbb{N}$ (Lemma 2.2.4). For any $m \in \mathsf{L}(\mathsf{Nat})$, $M \in \mathsf{Tm}_{\mathsf{Nat}}$ and $n \in \mathbb{N}$ we have that*

$$m\ \overline{\mathsf{eq}_{\mathsf{Nat}}}\ M \wedge m \Downarrow^{\forall} n \to M \Downarrow \underline{n}$$

Proof. The proof is by induction on $m \Downarrow^{\forall} n$. There are two cases to consider:

Case: $\eta^{\forall}(n) \Downarrow^{\forall} n$

By assumption $\eta^{\forall}(n)\ \overline{\mathsf{eq}_{\mathsf{Nat}}}\ M$ which by definition means that

$$\exists \underline{m}.(M \Downarrow \underline{m}) \wedge (n\, \mathsf{eq}_{\mathsf{Nat}}\, \underline{m})$$

But this implies $m = n$ and thus $M \Downarrow \underline{n}$.

Case: $\mathsf{step}^{\forall}(m) \Downarrow^{\forall} n$

By assumption we have $\mathsf{step}^{\forall}(m)\ \overline{\mathsf{eq}_{\mathsf{Nat}}}\ M$ and $\mathsf{step}^{\forall}(m) \Downarrow^{\forall} n$. Now Lemma 2.4.7 gives that $m\ \overline{\mathsf{eq}_{\mathsf{Nat}}}\ M$ and by the definition of $\Downarrow^{\forall}$, we have that $m \Downarrow^{\forall} n$. We can now use the inductive hypothesis to get $M \Downarrow \underline{n}$ □

2.5 A logical relation for contextual refinement

In this section, we define a logical relation by guarded recursion. It can be lifted to a coinductive relation, which we will show to imply contextual refinement.

Definition 2.5.1. We define the following relation by guarded recursion and induction on well-typed values.

$$\boxed{\preceq_{\sigma}^{\kappa,\mathsf{Val}} : [\![\sigma]\!]^{\kappa} \to \mathsf{Val}_{\sigma} \to \mathsf{Prop}} \qquad \boxed{\preceq_{\sigma}^{\kappa,\mathsf{Tm}} : \mathsf{L}^{\kappa}[\![\sigma]\!]^{\kappa} \to \mathsf{Tm}_{\sigma} \to \mathsf{Prop}}$$

$$\frac{}{n \preceq_{\mathsf{Nat}}^{\kappa,\mathsf{Val}} \underline{n}} \qquad \frac{}{\star \preceq_{1}^{\kappa,\mathsf{Val}} \langle\rangle} \qquad \frac{v \preceq_{\sigma}^{\kappa,\mathsf{Val}} V \qquad w \preceq_{\tau}^{\kappa,\mathsf{Val}} W}{(v,w) \preceq_{\sigma\times\tau}^{\kappa,\mathsf{Val}} \langle V, W\rangle} \qquad \frac{v \preceq_{\sigma}^{\kappa,\mathsf{Val}} V}{\mathbf{inl}\, v \preceq_{\sigma+\tau}^{\kappa,\mathsf{Val}} \mathsf{inl} V}$$

$$\frac{v \preceq^{\kappa,\mathsf{Val}}_{\tau} V}{\mathbf{inr}\, v \preceq^{\kappa,\mathsf{Val}}_{\sigma+\tau} \mathsf{inr} V} \qquad \frac{\forall w, W. w \preceq^{\kappa,\mathsf{Val}}_{\sigma} W \to v(w) \preceq^{\kappa,\mathsf{Tm}}_{\tau} M[W/x]}{v \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\, x.M}$$

$$\frac{\triangleright(\alpha : \kappa).(v[\alpha] \preceq^{\kappa,\mathsf{Val}}_{\tau[\mu X.\tau/X]} V)}{v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} \mathsf{fold} V} \qquad \frac{}{d \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M \triangleq d \; \overline{\preceq^{\kappa,\mathsf{Val}}_{\sigma}} \; M}$$

Similar to Lemma 2.3.7 and Lemma 2.4.6, we can prove that $- \preceq^{\kappa,\mathsf{Val}}_{\sigma} -$ and $- \preceq^{\kappa,\mathsf{Tm}}_{\sigma} -$ are guarded fixpoints and consequently, we also get coinductive liftings $- \preceq^{\mathsf{Val}}_{\sigma} - : [\![\sigma]\!] \times \mathsf{Val}_{\sigma} \to \mathsf{Prop}$ as well as $- \preceq_{\sigma} - : \forall\kappa.\mathsf{L}^{\kappa}[\![\sigma]\!]^{\kappa} \times \mathsf{Tm}_{\sigma} \to \mathsf{Prop}$ defined by

$$v \preceq^{\mathsf{Val}}_{\sigma} V \triangleq \forall\kappa.(v[\kappa] \preceq^{\kappa,\mathsf{Val}}_{\sigma} V)$$
$$d \preceq_{\sigma} M \triangleq \forall\kappa.(d[\kappa] \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M)$$

Compatibility Lemmas In preparation for the *guarded fundamental theorem* and *guarded congruence theorem*, we prove several lemmas which show that $\preceq^{\kappa,\mathsf{Tm}}_{\sigma}$ is compatible with the typing rules in Section 2.2.

Note, that the compatibility theorems for the unary constructors suc, pred, fst, snd, inl, inr, fold and unfold are analogous. We thus only prove the cases for pred, fold and unfold.

Lemma 2.5.2. *For all $d \in \mathsf{L}^{\kappa}\mathbb{N}$ and $M \in \mathsf{Tm}_{\mathsf{Nat}}$ such that $d \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} M$ we have that $\mathsf{L}^{\kappa}(\mathbf{suc})(d) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} \mathsf{suc} M$*

Proof. See Lemma 2.5.3. □

Lemma 2.5.3. *For all $d \in \mathsf{L}^{\kappa}\mathbb{N}$ and $M \in \mathsf{Tm}_{\mathsf{Nat}}$ such that $d \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} M$ we have that $\mathsf{L}^{\kappa}(\mathbf{pred})(d) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} \mathsf{suc} M$*

Proof. We apply Corollary 2.4.5 to $E = \mathsf{pred}([\,])$ and $f = \mathbf{pred}$. Now, if $n \preceq^{\kappa,\mathsf{Val}}_{\mathsf{Nat}} \underline{n}$, then $\mathsf{pred}\,\underline{n} \Downarrow \underline{\mathbf{max}(0, n-1)}$. Furthermore, we have that $\mathbf{pred}(n) \preceq^{\kappa,\mathsf{Val}}_{\mathsf{Nat}} \underline{\mathbf{max}(0, n-1)}$, and the claim follows. □

Lemma 2.5.4. *Assume that $d \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} L$, $e_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M$ and $e_2 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N$. Then also*

$$\left(d >>=^{\kappa} \begin{cases} 0 & \mapsto e_1 \\ n+1 & \mapsto e_2 \end{cases}\right) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \mathsf{ifz}(L, M, N)$$

Proof. We apply Lemma 2.4.4 to $E = \mathsf{ifz}([\,], M, N)$ and

$$f = \left(\lambda n. \begin{cases} n = 0 & \mapsto e_1 \\ n = m+1 & \mapsto e_2 \end{cases}\right).$$

It now suffices to prove that whenever $n \preceq^{\kappa,\mathsf{Val}}_{\mathsf{Nat}} \underline{n}$, we also have that $f(n) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} E[\underline{n}]$. If $n = 0$, then $(f(n) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} E[\underline{0}]) \simeq (e_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} E[\underline{0}])$. It follows from Lemma 2.4.3 that $E[\underline{0}] \rightsquigarrow^* M$ and by assumption we have $e_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M$. Thus, the claim follows.

For $n = m+1$ the reasoning is completely analogous. □

Lemma 2.5.5. *Assume that $d \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} L$, $e_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M$ and $e_2 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N$. Then also Let $d \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M$, then $\mathsf{L}^{\kappa}(\mathbf{inl})(d) \preceq^{\kappa,\mathsf{Tm}}_{\sigma+\tau} \mathsf{inl}\, M$*

Proof. Apply Corollary 2.4.5 to $E = \mathsf{inl}([\ ])$ and $f = \mathbf{inl}$. □

Lemma 2.5.6. *Assume that $d \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} L$, $e_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M$ and $e_2 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N$. Then also Let $d \preceq^{\kappa,\mathsf{Tm}}_{\tau} M$, then $\mathsf{L}^{\kappa}(\mathbf{inl})(d) \preceq^{\kappa,\mathsf{Tm}}_{\sigma+\tau} \mathsf{inr}\, M$*

Proof. Apply Corollary 2.4.5 to $E = \mathsf{inr}([\ ])$ and $f = \mathbf{inr}$. □

Lemma 2.5.7. *Assume*

1. $d \preceq^{\kappa,\mathsf{Tm}}_{\sigma_1+\sigma_2} L$
2. $\triangleright(\alpha : \kappa).(\forall v, V.v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1} V \to e_1[\alpha](v) \preceq^{\kappa,\mathsf{Tm}}_{\tau} M[V/x])$
3. $\triangleright(\alpha : \kappa).(\forall v, V.v \preceq^{\kappa,\mathsf{Val}}_{\sigma_2} V \to e_2[\alpha](v) \preceq^{\kappa,\mathsf{Tm}}_{\tau} N[V/x])$

then also

$$\left(d >\!\!>\!\!=^{\kappa} \begin{cases} \mathbf{inl}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).e_1[\alpha](v)) \\ \mathbf{inr}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).e_2[\alpha](v)) \end{cases}\right) \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{case}(L, x.M, y.N)$$

Proof. We apply Lemma 2.4.4 to $E = \mathsf{case}([\ ], x.M, y.N)$ and

$$f = \left(\lambda v. \begin{cases} \mathbf{inl}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).e_1[\alpha](v)) \\ \mathbf{inr}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).e_2[\alpha](v)) \end{cases}\right).$$

It now remains to show that if $v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} V$, then also $f(v) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} E[V]$. By definition of $v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} V$ there are two cases to consider: Either $\mathbf{inl}\, w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} \mathsf{inl}\, W$ or $\mathbf{inr}\, w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} \mathsf{inr}\, W$.

If $\mathbf{inl}\, w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} \mathsf{inl}\, W$, then $E[\mathsf{inl}\, W] \rightsquigarrow M[W/x]$ and thus Lemma 2.4.2 and Lemma 2.4.3 imply that

$$\begin{aligned}(f(\mathbf{inl}\, w) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} E[\mathsf{inl}\, W]) &\simeq \left(\mathsf{step}^{\kappa}\lambda(\alpha:\kappa).e_1[\alpha](w) \preceq^{\kappa,\mathsf{Tm}}_{\tau} E[\mathsf{inl}\, W]\right) \\ &\simeq \triangleright(\alpha:\kappa).\left(e_1[\alpha](w) \preceq^{\kappa,\mathsf{Tm}}_{\tau} E[\mathsf{inl}\, W]\right) \\ &\simeq \triangleright(\alpha:\kappa).\left(e_1[\alpha](w) \preceq^{\kappa,\mathsf{Tm}}_{\tau} M[W/x]\right),\end{aligned}$$

which is true by assumption.

The case where $\mathbf{inr}\, w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} \mathsf{inr}\, W$ is completely analogous □

Lemma 2.5.8. *For $d \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M$ and $e \preceq^{\kappa,\mathsf{Tm}}_{\tau} N$ we have*

$$(d >>=^{\kappa} \lambda v.e >>=^{\kappa} \lambda w.\eta^{\kappa}(v,w)) \preceq^{\kappa,\mathsf{Tm}}_{\sigma\times\tau} \langle M,N\rangle$$

Proof. We apply Lemma 2.4.4 iteratively. First, let $E = \langle [\,], N\rangle$ and

$$f = (\lambda v.(e >>=^{\kappa} \lambda w.\eta^{\kappa}(v,w)))\,.$$

We have to show that $v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V$ we have $f(v) \preceq^{\kappa,\mathsf{Tm}}_{\sigma\times\tau} E[V]$. This is equivalent to proving that $(e >>=^{\kappa} \lambda w.\eta^{\kappa}(v,w)) \preceq^{\kappa,\mathsf{Tm}}_{\sigma\times\tau} \langle V,N\rangle$.

To show this, we apply Lemma 2.4.4 again, now to $E = \langle V, [\,]\rangle$ and $g = \lambda w.\eta^{\kappa}(v,w)$. Now it remains to prove that if $w \preceq^{\kappa,\mathsf{Val}}_{\tau} W$, then also $g(w) \preceq^{\kappa,\mathsf{Tm}}_{\sigma\times\tau} \langle V,W\rangle$. But since $g(w) = \eta^{\kappa}(v,w)$ and $\langle V,W\rangle$ is a value, this is equivalent to $(v,w) \preceq^{\kappa,\mathsf{Val}}_{\sigma\times\tau} \langle V,W\rangle$, which by definition follows from the assumptions $v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V$ and $w \preceq^{\kappa,\mathsf{Val}}_{\tau} W$. □

Lemma 2.5.9. *If $d \preceq^{\kappa,\mathsf{Tm}}_{\sigma\times\tau} M$, then $\mathsf{L}^{\kappa}(\mathsf{pr}_1)(d) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \mathsf{fst}\,M$*

Proof. Apply Corollary 2.4.5 to $E = \mathsf{fst}\,([\,])$ and $f = \mathsf{pr}_1$. □

Lemma 2.5.10. *If $d \preceq^{\kappa,\mathsf{Tm}}_{\sigma\times\tau} M$, then $\mathsf{L}^{\kappa}(\mathsf{pr}_2)(d) \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{snd}\,M$*

Proof. Apply Corollary 2.4.5 to $E = \mathsf{snd}\,([\,])$ and $f = \mathsf{pr}_2$. □

Lemma 2.5.11. *If $d \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} M$ and $e \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N$, then $d \cdot e \preceq^{\kappa,\mathsf{Tm}}_{\tau} MN$*

Proof. We apply Lemma 2.4.4 iteratively and start with $E = [\,]N$ and $f = \lambda v'.(e >>=^{\kappa} \lambda w.\mathsf{step}^{\kappa}(\mathsf{next}^{\kappa}(v'w)))$. Now, we have to show that $v \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\,x.M$ implies that $f(v) \preceq^{\kappa,\mathsf{Tm}}_{\tau} E[\mathsf{lam}\,x.M]$.
This is equivalent to $e >>=^{\kappa} \lambda w.\mathsf{step}^{\kappa}(\mathsf{next}^{\kappa}(vw)) \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{lam}\,x.M(N)$, which follows by Lemma 2.4.3 and the fact that $E[\mathsf{lam}\,x.M] \rightsquigarrow \mathsf{lam}\,x.M(N)$.

To prove this, we apply Lemma 2.4.4 again, this time to $E = V[\,]$ and $g = \lambda w'.\mathsf{step}^{\kappa}(\mathsf{next}^{\kappa}(vw'))$. Now it remains to show that if $w \preceq^{\kappa,\mathsf{Val}}_{\sigma} W$, then also $g(w) \preceq^{\kappa,\mathsf{Tm}}_{\tau} E[W]$. The latter is equivalent to $\mathsf{step}^{\kappa}(\mathsf{next}^{\kappa}(vw)) \preceq^{\kappa,\mathsf{Tm}}_{\tau} M[W/x]$, since $E[W] \rightsquigarrow M[W/x]$.

But this follows from the definition of $v \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\,x.M$ and $w \preceq^{\kappa,\mathsf{Val}}_{\sigma} W$. □

Lemma 2.5.12. *If $d \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} M$ then also $\mathsf{L}^{\kappa}(\mathsf{next}^{\kappa})(d) \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} \mathsf{fold}\,M$.*

Proof. We apply Corollary 2.4.5 to $E = \mathsf{fold}\,[\,]$ and $f = \mathsf{next}^{\kappa}$. It remains to show that if $v \preceq^{\kappa,\mathsf{Val}}_{\tau[\mu X.\tau/X]} V$, then also $f(v) \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} E[V]$. Since $E[V] = \mathsf{fold}\,V$, this is equivalent to $\triangleright(\alpha:\kappa).\left((\lambda(\alpha:\kappa).v)\,[\alpha] \preceq^{\kappa,\mathsf{Val}}_{\tau[\mu X.\tau/X]} V\right)$, which follows by definition of $\preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau}$. □

Lemma 2.5.13. *If $d \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} M$ then*

$$d >>=^{\kappa} \lambda v.\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\eta^{\kappa}(v[\alpha])) \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} \mathsf{unfold}\,M.$$

Proof. We apply Lemma 2.4.4 to $E = \mathsf{unfold}([\,])$ and

$$f = \lambda v.\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\eta^{\kappa}(v[\alpha])).$$

Now it remains to show that for $v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} \mathsf{fold}\, V$ we have that $f(v) \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} E[\mathsf{fold}\, V]$. The latter is equivalent to $\mathsf{step}^{\kappa}\lambda(\alpha:\kappa).\eta^{\kappa}(v[\alpha]) \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} V$, by Lemma 2.4.3 and the fact that $E[\mathsf{fold}\, V] \rightsquigarrow V$.

Applying Lemma 2.4.2 and Lemma 2.4.1 implies that we only have to prove that $\triangleright(\alpha:\kappa).\left(v[\alpha] \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} V\right)$. However, this follows from the assumption $v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} \mathsf{fold}\, V$ by the definition of $\preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau}$. □

Fundamental lemma

For our logical relation to imply contextual refinement, we need it to validate two quintessential properties: The *fundamental lemma* and the *congruence lemma*. Here we prove that for both, the guarded as well as the coinductive logical relation, the fundamental lemma is true. As a direct consequence, we see that denotationally equal terms are in the relation.

In the following, we also consider open terms. The interpretation of open terms requires an environment $\rho : [\![\Gamma]\!]^{\kappa}$ and the operational semantics only applies to closed terms. Therefore, for open terms M and N the proposition $[\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta]$ is only meaningful for a suitable pair of environments ρ and closing substitutions δ. In general, $[\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta]$ can only be true if ρ and δ are related in some sense. Before we state the fundamental lemma, we define which environment and closing substitution pairs have to be considered.

Definition 2.5.14. For $\delta : \mathsf{Sub}(\cdot;\Gamma)$ and $\rho \in [\![\Gamma]\!]^{\kappa}$ we define

$$\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta \triangleq \forall(x:\sigma) \in \Gamma.\rho(x) \preceq^{\kappa,\mathsf{Val}}_{\sigma} \delta(x)$$

$\delta : \mathsf{Sub}(\cdot;\Gamma)$ is the precise way of denoting a closing substitution for Γ — it contains a closed value for every variable in Γ.

Lemma 2.5.15 (Guarded Fundamental Lemma)**.** *For all $M \in \mathsf{Tm}^{\Gamma}_{\sigma}$ we have that* $\forall\rho,\delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M[\delta]$

Proof. The proof proceeds by induction on $M : \mathsf{Tm}^{\Gamma}_{\sigma}$ and relies almost entirely on the compatibility lemmas with the only exception of the $M = \mathsf{lam}\, x.M'$ case.

Case: $\mathsf{lam}\, x.M'$

By the induction hypothesis, we have that

$$\forall\rho,\delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma,\sigma} \delta) \to [\![M']\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau} M'[\delta].$$

Thus, we get that

$$\forall\rho,\delta,v,V.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \wedge (v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V) \to [\![M']\!]^{\kappa}_{\rho,v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} M'[\delta, V/x]$$

$$= \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V) \to (\llbracket M' \rrbracket^{\kappa}_{\rho,v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} M'[\delta, V/x])$$
$$= \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \llbracket \mathsf{lam}\ x.M' \rrbracket^{\mathsf{Val},\kappa}_{\rho} \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\ x.M'[\delta]$$
$$= \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \eta^{\kappa}(\llbracket \mathsf{lam}\ x.M' \rrbracket^{\mathsf{Val},\kappa}_{\rho}) \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} \mathsf{lam}\ x.M'[\delta]$$
$$= \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \llbracket \mathsf{lam}\ x.M' \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} \mathsf{lam}\ x.M'[\delta]$$

Case: MN
Let $\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta$, by induction hypothesis we have $\llbracket M \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} M[\delta]$ and $\llbracket N \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta]$. Now Lemma 2.5.11 implies that $\llbracket MN \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau} MN[\delta]$.

Case: $\mathsf{case}(L, x.M, y.N)$
Let $\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta$, by induction hypothesis we have $\llbracket L \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma_1+\sigma_2} L[\delta]$. Furthermore, it follows from the induction hypothesis that

$$\forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1} V) \to \llbracket M \rrbracket^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} M[\delta, V/x]$$

as well as

$$\forall w, W.(w \preceq^{\kappa,\mathsf{Val}}_{\sigma_2} W) \to \llbracket N \rrbracket^{\kappa}_{\rho.w} \preceq^{\kappa,\mathsf{Tm}}_{\tau} N[\delta, V/y].$$

But now we can simply apply Lemma 2.5.7 to get

$$\left(\llbracket L \rrbracket^{\kappa}_{\rho} >\!\!>\!\!=^{\kappa} \begin{cases} \mathbf{inl}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\llbracket M \rrbracket^{\kappa}_{\rho.v}) \\ \mathbf{inr}\, w \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\llbracket N \rrbracket^{\kappa}_{\rho.w}) \end{cases} \right) \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{case}(L, x.M, y.N)[\delta]$$

Case: $\mathsf{unfold}\, M$
Let $\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta$, by the induction hypothesis we have $\llbracket M \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} M[\delta]$ and now Lemma 2.5.13 implies that $\llbracket \mathsf{unfold}\, M \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau\mu X.\tau/X} \mathsf{unfold}\, M[\delta]$
The remaining cases follow similarly. □

Corollary 2.5.16. *For terms $M, N : \mathsf{Tm}^{\Gamma}_{\sigma}$ we have that*

$$(\forall \rho : \llbracket \Gamma \rrbracket^{\kappa}. \llbracket M \rrbracket^{\kappa}_{\rho} = \llbracket N \rrbracket^{\kappa}_{\rho}) \to \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \llbracket M \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta] \right).$$

Proof. The fundamental lemma implies that $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \llbracket N \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta]$. But now we can just replace $\llbracket N \rrbracket^{\kappa}_{\rho}$ by $\llbracket M \rrbracket^{\kappa}_{\rho}$ and the claim follows. □

Lemma 2.5.17 (Coinductive Fundamental Lemma)**.** *Let $M \in \mathsf{Tm}^{\Gamma}_{\sigma}$, then*

$$\forall \kappa.(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \llbracket M \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M[\delta])$$

Proof. Let $M \in \mathsf{Tm}^{\Gamma}_{\sigma}$, then by Lemma 2.5.15 we get that $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \llbracket M \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M[\delta]$ and consequently also

$$\forall \kappa.(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \llbracket M \rrbracket^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M[\delta]).$$

□

Congruence lemma

We now turn to the other quintessential property for logical relations: The *congruence lemma*. It says that the logical relation is closed under typed contexts $C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$. We prove a congruence lemma for the guarded as well as the coinductive relation.

Lemma 2.5.18 (Guarded Congruence Lemma)**.** *For any terms $M, N \in \mathsf{Tm}^{\Gamma}_{\sigma}$ and every context $C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$ we get that if $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta]$ then also $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau} (C[N])[\delta]$*

Proof. The proof proceeds by induction on context derivations. All cases — except for the $M = \mathsf{lam}\, x.M'$ case — are direct consequences of the compatibility lemmas.

Case: $C = \mathsf{lam}\, x.C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau_1 \to \tau_2)$
Assume $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta]$. We have that $C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta, (x : \tau_1) \vdash \tau_2)$ and by the induction hypothesis it follows that

$$\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta, x:\tau_1} \delta) \to [\![C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau_2} (C'[N])[\delta]$$

It now follows that

$$\begin{aligned}
&\forall \rho, \delta.\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta,\tau_1} \delta \to [\![C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau_2} (C'[N])[\delta] \\
&= \forall \rho, \delta, v, V.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta \wedge v \preceq^{\kappa,\mathsf{Val}}_{\tau_1} V) \to [\![C'[M]]\!]^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau_2} (C'[N])[\delta, V/x] \\
&= \forall \rho, \delta(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to \left(\forall v, V.v \preceq^{\kappa,\mathsf{Val}}_{\tau_1} V \to [\![C'[M]]\!]^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau_2} (C'[N])[\delta, V/x]\right) \\
&= \forall \rho, \delta(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![\mathsf{lam}\, x.C'[M]]\!]^{\mathsf{Val},\kappa}_{\rho} \preceq^{\kappa,\mathsf{Val}}_{\tau_1 \to \tau_2} (\mathsf{lam}\, x.C'[N])[\delta, V/x] \\
&= \forall \rho, \delta(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to \eta^{\kappa}([\![\mathsf{lam}\, x.C'[M]]\!]^{\mathsf{Val},\kappa}_{\rho}) \preceq^{\kappa,\mathsf{Tm}}_{\tau_1 \to \tau_2} (\mathsf{lam}\, x.C'[N])[\delta, V/x] \\
&= \forall \rho, \delta(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau_1 \to \tau_2} (C[N])[\delta, V/x]
\end{aligned}$$

Case: $C = \mathsf{fold} C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mu X.\tau)$
Assume $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta]$. We have that $C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau[\mu X.\tau/X])$ and thus by induction hypothesis we have

$$\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} (C'[N])[\delta]$$

By Lemma 2.5.12 we have that

$$\begin{aligned}
&\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} (C'[N])[\delta] \\
&\to \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C'[M]]\!]^{\kappa}_{\rho} >>=^{\kappa} \lambda v.\eta^{\kappa}(\lambda(\alpha : \kappa).v) \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} (\mathsf{fold} C'[N])[\delta]\right) \\
&= \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![\mathsf{fold} C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} (\mathsf{fold} C'[N])[\delta]
\end{aligned}$$

Case: $C = \mathsf{case}(L, x.C', y.N) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$
Assume that $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} M'[\delta]$. We furthermore have that

1. $\Delta \vdash L : \tau_1 + \tau_2$
2. $C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta, x : \tau_1 \vdash \tau)$
3. $\Delta, y : \tau_2 \vdash N : \tau$

By the induction hypothesis, we have

$$\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta,\tau_1} \delta) \to [\![C'[M]]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau (C'[M'])\delta.$$

Furthermore, Lemma 2.5.15 implies that $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Delta \delta) \to [\![L]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_{\tau_1+\tau_2} L[\delta]$ and $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta,\tau_1} \delta) \to [\![N]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau N[\delta]$.

Thus, for all ρ and δ such that $\rho \preceq^{\kappa,\mathsf{Val}}_\Delta \delta$, we get that

1. $[\![L]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_{\tau_1+\tau_2} L[\delta]$
2. $\forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\tau_1} V) \to [\![C'[M]]\!]^\kappa_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_\tau (C'[M'])[\delta, V/x]$
3. $\forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\tau_2} V) \to [\![N]\!]^\kappa_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_\tau N[\delta, V/x]$

Now, using Lemma 2.5.7 we conclude that

$$[\![L]\!]^\kappa_\rho >>=^\kappa \begin{cases} \mathbf{inl}\, v \mapsto \mathsf{step}^\kappa(\lambda(\alpha : \kappa).[\![C'[M]]\!]^\kappa_{\rho.v}) \\ \mathbf{inr}\, v \mapsto \mathsf{step}^\kappa(\lambda(\alpha : \kappa).[\![N]\!]^\kappa_{\rho.v}) \end{cases} \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{case}(L, x.C'[M'], y.N)[\delta]$$

and thus we get that

$$\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Delta \delta) \to [\![C[M]]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau C[M']$$

The remaining cases are similar. □

Theorem 2.5.19 (Coinductive Congruence Theorem)**.** *For any terms $M, N \in \mathsf{Tm}^\Gamma_\sigma$ and every context $C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$, if $\forall \kappa. \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Gamma \delta) \to [\![M]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\sigma N[\delta]\right)$, then also*

$$\forall \kappa. \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Delta \delta) \to [\![C[M]]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau (C[N])[\delta]\right)$$

Proof. Assume $\forall \kappa. \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Gamma \delta) \to [\![M]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\sigma N[\delta]\right)$. We want to show $\forall \kappa. \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Delta \delta) \to [\![C[M]]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau (C[N])[\delta]\right)$, and thus let $\kappa : \mathscr{C}$, by clock application we get that $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Gamma \delta) \to [\![M]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\sigma N[\delta]$.

We now apply Lemma 2.5.18 and get $\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Delta \delta) \to [\![C[M]]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau (C[N])[\delta]$. □

Contextual refinement

Finally, we can put all the pieces together and show that the logical relation implies contextual refinement.

Theorem 2.5.20. *Let* $M,N : \mathsf{Tm}^{\Gamma}_{\sigma}$, *then*

$$\left(\forall\kappa.(\forall\rho,\delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta])\right) \to M \preceq_{\mathsf{Ctx}} N$$

Proof. Let $M,N : \mathsf{Tm}^{\Gamma}_{\sigma}$ be such that $\forall\kappa.(\forall\rho,\delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} N[\delta])$. We have to show that for any $C : (\Gamma \vdash \sigma) \Rightarrow (\vdash \mathsf{Nat})$ it is the case that

$$C[M] \Downarrow \underline{n} \to C[N] \Downarrow \underline{n}.$$

Let $C : (\Gamma \vdash \sigma) \Rightarrow (\vdash \mathsf{Nat})$ be arbitrary, by Theorem 2.5.19 it follows that

$$\forall\kappa.([\![C[M]]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} C[N]).$$

Now assume that $C[M] \Downarrow \underline{n}$, then by Theorem 2.3.9 we have that $[\![C[M]]\!] \Downarrow^{\forall} n$.

Thus, Lemma 2.4.8 implies that $C[N] \Downarrow \underline{n}$. Note that Lemma 2.4.8 applies, since $\forall\kappa.(\preceq^{\kappa,\mathsf{Val}}_{\mathsf{Nat}}) = \mathsf{eq}_{\mathsf{Nat}}$. □

2.6 Examples

We now give additional examples to showcase the usefulness of the denotational semantics and the logical relation.

Example 21. For any term $\Gamma \vdash M : \tau[\mu X.\tau/X]$ and any $\rho : [\![\Gamma]\!]^{\kappa}$ it is the case that

$$[\![\mathsf{unfold}\,(\mathsf{fold}\,M)]\!]^{\kappa}_{\rho} = \mathsf{step}^{\kappa}(\lambda(\alpha : \kappa).[\![M]\!]^{\kappa}_{\rho})$$

Proof. We have that

$$\begin{aligned}[\![\mathsf{unfold}\,(\mathsf{fold}\,M)]\!]^{\kappa}_{\rho} &= ([\![\mathsf{fold}\,M]\!]^{\kappa}_{\rho}) >>=^{\kappa} \lambda v.\mathsf{step}^{\kappa}(\lambda\alpha.\eta^{\kappa}(v[\alpha])) \\ &= \mathsf{L}^{\kappa}(\mathsf{next}^{\kappa})([\![M]\!]^{\kappa}_{\rho}) >>=^{\kappa} \lambda v.\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\eta^{\kappa}(v[\alpha]))\end{aligned}$$

Since $\mathsf{L}^{\kappa}(\mathsf{next}^{\kappa})$ is a delay algebra homomorphism, it commutes with $>>=^{\kappa}$ and consequently we get

$$\begin{aligned}&= [\![M]\!]^{\kappa}_{\rho} >>=^{\kappa} \lambda v.\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\eta^{\kappa}(\mathsf{next}^{\kappa}(v)[\alpha])) \\ &= [\![M]\!]^{\kappa}_{\rho} >>=^{\kappa} \lambda v.(\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\eta^{\kappa}v))\end{aligned}$$

Since step^{κ} is a delay algebra homomorphism, we furthermore get that

$$\begin{aligned}&= \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).([\![M]\!]^{\kappa}_{\rho} >>=^{\kappa} \lambda v.\eta^{\kappa}v)) \\ &= \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).([\![M]\!]^{\kappa}_{\rho}))\end{aligned}$$

□

The following example shows that $\bot^\kappa$ refines every program.

Example 22. Let $\bot^\kappa_\sigma : \mathsf{L}^\kappa[\![\sigma]\!]^\kappa$ be defined by $\bot^\kappa_\sigma \triangleq \mathsf{fix}^\kappa \mathsf{step}^\kappa$.

For any well-typed term M we have that

$$(\bot^\kappa_\sigma \preceq^{\kappa,\mathsf{Tm}}_\sigma M) = \triangleright^\kappa(\bot^\kappa_\sigma \preceq^{\kappa,\mathsf{Tm}}_\sigma M)$$

Consequently, for any well-typed term M we have $\bot^\kappa_\sigma \preceq^{\kappa,\mathsf{Tm}}_\sigma M$

We now turn to the previous example of recursive functions, and show that $(\mathsf{Y}f)V$ and $f(\mathsf{Y}f)V$ are contextually equivalent for any value V. Note that in a call-by-value setting, this is the correct formulation of this theorem: $\mathsf{Y}f$ and $f(\mathsf{Y}f)$ are not necessarily contextually equivalent.

Theorem 2.6.1. *For any open value* $\Gamma \vdash f : (\sigma \to \tau) \to \sigma \to \tau$ *and open value* $\Gamma \vdash V : \sigma$ *it is the case that*

$$(\mathsf{Y}f)V \preceq_{\mathsf{Ctx}} f(\mathsf{Y}f)V$$

as well as

$$f(\mathsf{Y}f)V \preceq_{\mathsf{Ctx}} (\mathsf{Y}f)V$$

Proof. Both directions are completely analogous, we thus only show one of them. The fundamental theorem Lemma 2.5.17 directly implies that both

$$\forall \kappa.\left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Gamma \delta) \to [\![(\mathsf{Y}f)V]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau ((\mathsf{Y}f)V)[\delta]\right)$$

From Example 17 we know that $((\mathsf{Y}f)[\delta])(V[\delta]) \rightsquigarrow^* ((f(\lambda z.e_f(\mathsf{fold}\, e_f)z))[\delta])(V[\delta])$ and also $(f(\mathsf{Y}f)[\delta])(V[\delta]) \rightsquigarrow^* ((f(\lambda z.e_f(\mathsf{fold}\, e_f)z))[\delta])(V[\delta])$.

We can use Lemma 2.4.3 iteratively to get that

$$\forall \kappa.\left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_\Gamma \delta) \to [\![(\mathsf{Y}f)V]\!]^\kappa_\rho \preceq^{\kappa,\mathsf{Tm}}_\tau ((f(\mathsf{Y}f))V)[\delta]\right)$$

But now Theorem 2.5.20 directly implies that $(Yf)V \preceq_{\mathsf{Ctx}} f(Yf)V$. □

In the following, we investigate a more complicated example that involves fixpoints and higher-order functions.

Syntactic Minimal Invariance Let $\tau \triangleq \mu X.1 + X \to X$.

It is easy to see that a function

$$\mathsf{id}_{\mathsf{alt}} \triangleq \mathsf{lam}\, x.\mathsf{case}(\mathsf{unfold}\, x, y.\mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle)), g.\mathsf{fold}\,(\mathsf{inr}\,(g)))$$

should be contextually equivalent to the identity function. Note, that either side of the case elimination is basically the identity function (modulo the folding and unfolding of the recursive type τ).

Our goal is to prove that the identity function id is contextually equivalent to the recursively defined function h defined in Fig. 2.7. This means, that we have to prove

$$
\begin{array}{l}
h \triangleq \mathsf{Y}(F) \text{ where} \\
\quad F : (\tau \to \tau) \to \tau \to \tau \\
\quad F \triangleq \mathsf{lam}\ f.(\mathsf{lam}(x:\tau).\ \mathsf{let}\ (t = \mathsf{unfold}\, x)\ \mathsf{in} \\
\qquad\qquad \mathsf{case}(t, y.\mathsf{case1}(y), g.\mathsf{case2}(g))),\ \text{where} \\
\qquad\qquad\quad \mathsf{case1}(-) \triangleq \mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle)) \\
\qquad\qquad\quad \mathsf{case2}(g) \triangleq \mathsf{fold}\,(\mathsf{inr}\,(\mathsf{lam}\ y.f(g(f(y)))))) \\
h_{val} \triangleq \mathsf{lam}\ z.e_F(\mathsf{fold}\,(e_F))z,\ \text{where} \\
\quad e_F : (\mu X.X \to \tau \to \tau) \to \tau \to \tau \\
\quad e_F \triangleq \mathsf{lam}\ y.\mathsf{let}\ (y' = \mathsf{unfold}\, y)\ \mathsf{in}\ F(\mathsf{lam}\ x.y'yx)
\end{array}
$$

Figure 2.7: Definition of recursive identity

$\mathsf{id} \preceq_{\mathsf{Ctx}} h$ and $h \preceq_{\mathsf{Ctx}} \mathsf{id}$. The function $h : \tau \to \tau$ complicates the function $\mathsf{id}_{\mathsf{alt}}$ by interleaving the second case with recursive calls of a function variable. The challenge is to realize that by definition the recursive call of f is forced to be the identity.

h_{val}, inlcase and inrcase are all of type $\tau \to \tau$. We have that $h \Downarrow h_{val}$ and furthermore, there exists a value $V_{F(h)}$ such that $F(h) \Downarrow V_{F(h)}$ (in either case just one reduction step).

We start by proving $\mathsf{id} \preceq_{\mathsf{Ctx}} h$ To do so, we use Theorem 2.5.20. It suffices to show that $\lambda x.x \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} h_{val}$, since by a simple calculation we get

$$[\![\mathsf{id}]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{\tau\to\tau} h \simeq \lambda x.x \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} h_{val}$$

We proceed by guarded recursion and thus prove

$$\triangleright^{\kappa}(\lambda x.x \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} h_{val}) \to \lambda x.x \preceq^{\kappa,\mathsf{Tm}}_{\tau\to\tau} h_{val}$$

Thus, assume the guarded hypothesis and let v, V be arbitrary such that $v \preceq^{\kappa,\mathsf{Val}}_{\tau} \mathsf{fold}\, V$ (all values of type τ are of the form $\mathsf{fold}\, V$). We ought to show that $v \preceq^{\kappa,\mathsf{Tm}}_{\tau} h_{val}(\mathsf{fold}\, V)$. However, by Lemma 2.4.3 it is sufficient to prove $v \preceq^{\kappa,\mathsf{Tm}}_{\tau} (F(h_{val}))(\mathsf{fold}\, V)$, since there exists a value $V_{\mathsf{Y}F}$ such that $(F(\mathsf{Y}F))V \rightsquigarrow^* V_{\mathsf{Y}F}$ as well as $(\mathsf{Y}F)(V) \rightsquigarrow^* V_{\mathsf{Y}F}$.

We proceed by distinguishing the different cases for $v \preceq^{\kappa,\mathsf{Val}}_{\tau} \mathsf{fold}\, V$. Initially, the only case to consider is $\triangleright(\alpha:\kappa).(v\,[\alpha] \preceq^{\kappa,\mathsf{Val}}_{1+\tau\to\tau} V)$. However, we furthermore distinguish the cases for $v\,[\alpha] \preceq^{\kappa,\mathsf{Val}}_{1+\tau\to\tau} V$, which leaves us with two cases:

1. $\triangleright(\alpha:\kappa).(\star \preceq^{\kappa,\mathsf{Val}}_{1} \langle\rangle)$ with $v\,[\alpha] = \mathbf{inl}\,(\star)$ and $V = \mathsf{inl}\,(\langle\rangle)$
2. $\triangleright(\alpha:\kappa).(w \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} W)$ with $v\,[\alpha] = \mathbf{inr}\,(w)$ and $V = \mathsf{inr}\,(W)$

Case: $v[\alpha] = \mathbf{inl}(\star)$ and $V = \mathsf{inl}(\langle\rangle)$
Since $(F(h_{val}))(\mathsf{fold}(\mathsf{inl}(\langle\rangle))) \Downarrow \mathsf{fold}(\mathsf{inl}(\langle\rangle))$, it follows that

$$\begin{aligned}\triangleright^\kappa(\star \preceq^{\kappa,\mathsf{Tm}}_1 \langle\rangle) &\simeq \triangleright(\alpha:\kappa).(v[\alpha] \preceq^{\kappa,\mathsf{Tm}}_{1+\tau\to\tau} \mathsf{inl}(\langle\rangle)) \\ &\simeq v \preceq^{\kappa,\mathsf{Tm}}_\tau (F(h_{val}))(\mathsf{fold}\, V)\end{aligned}$$

and the claim follows.

Case: $\triangleright(\alpha:\kappa).(w \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} W)$ with $v[\alpha] = \mathbf{inr}(w)$ and $V' = \mathsf{inr}(W)$
We have that $F(h_{val})(\mathsf{fold}(\mathsf{inr}(W))) \rightsquigarrow^* \mathsf{fold}(\mathsf{inr}(\mathsf{lam}\, y.h_{val}(W(h_{val}(y)))))$ and thus

$$\begin{aligned}&\triangleright^\kappa(w \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} \mathsf{lam}\, y.h_{val}(W(h_{val}(y)))) \\ \simeq\ &\triangleright^\kappa\left(\mathsf{inr}\, w \preceq^{\kappa,\mathsf{Val}}_{1+\tau\to\tau} \mathsf{inr}(\mathsf{lam}\, y.h_{val}(W(h_{val}(y))))\right) \\ \simeq\ &v \preceq^{\kappa,\mathsf{Val}}_\tau \mathsf{fold}(\mathsf{inr}(\mathsf{lam}\, y.h_{val}(W(h_{val}(y))))) \\ \simeq\ &(\lambda x.x)(v) \preceq^{\kappa,\mathsf{Tm}}_\tau F(h_{val})(V)\end{aligned}$$

It remains to show that $\triangleright^\kappa(w \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} \mathsf{lam}\, y.h_{val}(W(h_{val}(y))))$, which is by definition equivalent to

$$\triangleright^\kappa\left(\forall\tilde{v},\tilde{V}.(\tilde{v} \preceq^{\kappa,\mathsf{Val}}_\tau \tilde{V}) \to w(\tilde{v}) \preceq^{\kappa,\mathsf{Tm}}_\tau h_{val}(W(h_{val}(\tilde{V})))\right).$$

To prove this, we first assume a tick $\alpha:\kappa$. Under tick, we are free to apply the guarded hypothesis and thus the claim follows by applying Lemma 2.5.11 three times:

1. $\tilde{v} \preceq^{\kappa,\mathsf{Val}}_\tau \tilde{V}$ and the guarded hypothesis (under tick) $[\![\mathsf{id}]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\tau\to\tau} h_{val}$ give that $\tilde{v} \preceq^{\kappa,\mathsf{Tm}}_\tau h_{val}(\tilde{V})$
2. $\tilde{v} \preceq^{\kappa,\mathsf{Tm}}_\tau h_{val}(\tilde{V})$ and the assumption $w \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} W$ give $w(\tilde{v}) \preceq^{\kappa,\mathsf{Tm}}_\tau W(h_{val}(\tilde{V}))$
3. $w(\tilde{v}) \preceq^{\kappa,\mathsf{Tm}}_\tau W(h_{val}(\tilde{V}))$ and the guarded hypothesis $[\![\mathsf{id}]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\tau\to\tau} h_{val}$ give that $w(\tilde{v}) \preceq^{\kappa,\mathsf{Tm}}_\tau h_{val}(W(h_{val}(\tilde{V})))$.

This concludes the first direction.

The other direction $h \preceq_{\mathsf{Ctx}} \mathsf{id}$ Using Theorem 2.5.20, we only have to show that $[\![h]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\tau\to\tau} \mathsf{id}$. Similarly as before, we prove instead $[\![h_{val}]\!]^{\mathsf{Val},\kappa} \preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau} \mathsf{lam}\, x.x$ by guarded recursion. To that goal, we assume the guarded hypothesis and v, V arbitrary such that $v \preceq^{\kappa,\mathsf{Val}}_\tau \mathsf{fold}\, V$. We have to show that $[\![h_{val}]\!]^{\mathsf{Val},\kappa}(v) \preceq^{\kappa,\mathsf{Tm}}_\tau x[\mathsf{fold}\, V/x]$.

First, recall Example 20 and the definition of h_{val}. By a similar line of reasoning, we get

$$\begin{aligned}[\![h_{val}]\!]^{\mathsf{Val},\kappa}(v) &\triangleq [\![\mathsf{lam}\, z.e_F(\mathsf{fold}(e_F))z]\!]^{\mathsf{Val},\kappa}(v) \\ &= [\![e_F(\mathsf{fold}(e_F))z]\!]^\kappa_{z\mapsto v}\end{aligned}$$

$$
\begin{aligned}
&= (\Delta^\kappa)^3([\![F(\mathsf{lam}\ z.e_F(\mathsf{fold}\,(e_F))z)]\!]^\kappa \cdot v)\\
&= (\Delta^\kappa)^3([\![F(h_{val})]\!]^\kappa \cdot v)\\
&= (\Delta^\kappa)^5([\![V_{F(h)}]\!]^{\mathsf{Val},\kappa}(v))\\
&= (\Delta^\kappa)^6\,(\mathsf{step}^\kappa\,(\lambda\,(\alpha:\kappa).e_{\mathsf{case}}(v\,[\alpha]))\,,\ \text{where}\\
&\quad e_{case} \triangleq \begin{cases} \mathsf{inl}\,(v') \mapsto \Delta^\kappa[\![\mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle))]\!]^{\mathsf{Val},\kappa} \\ \mathsf{inr}\,(v') \mapsto \Delta^\kappa[\![\mathsf{fold}\,(\mathsf{inr}\,(\mathsf{lam}\ y.h_{val}(g(h_{val}(y)))))]\!]^{\mathsf{Val},\kappa} \end{cases}
\end{aligned}
$$

Now it suffices to prove that $\mathsf{step}^\kappa\,(\lambda\,(\alpha:\kappa).e_{\mathsf{case}}(v\,[\alpha]))\ \preceq_\tau^{\kappa,\mathsf{Tm}}\ \mathsf{fold}\,V$, since by the previous calculations it follows

$$
\begin{aligned}
&\mathsf{step}^\kappa\,(\lambda\,(\alpha:\kappa).e_{\mathsf{case}}(v\,[\alpha]))\ \preceq_\tau^{\kappa,\mathsf{Tm}}\ \mathsf{fold}\,V\\
\to\ &(\triangleright^\kappa)^6\left(\mathsf{step}^\kappa\,(\lambda\,(\alpha:\kappa).e_{\mathsf{case}}(v\,[\alpha]))\ \preceq_\tau^{\kappa,\mathsf{Tm}}\ \mathsf{fold}\,V\right)\\
\simeq\ &(\Delta^\kappa)^6\,(\mathsf{step}^\kappa\,(\lambda\,(\alpha:\kappa).e_{\mathsf{case}}(v\,[\alpha])))\ \preceq_\tau^{\kappa,\mathsf{Tm}}\ \mathsf{fold}\,V\\
\simeq\ &[\![h_{val}]\!]^{\mathsf{Val},\kappa}(v)\ \preceq_\tau^{\kappa,\mathsf{Val}}\ \mathsf{fold}\,V
\end{aligned}
$$

As for the previous direction, we proceed by inducting on $v\ \preceq_\tau^{\kappa,\mathsf{Val}}\ \mathsf{fold}\,V$, which leaves us with two cases:

1. $\triangleright\,(\alpha:\kappa).(\star\ \preceq_1^{\kappa,\mathsf{Val}}\ \langle\rangle)$ with $v\,[\alpha] = \mathbf{inl}\,(\star)$ and $V = \mathsf{inl}\,(\langle\rangle)$
2. $\triangleright\,(\alpha:\kappa).(w\ \preceq_{\tau\to\tau}^{\kappa,\mathsf{Val}}\ W)$ with $v\,[\alpha] = \mathbf{inr}\,(w)$ and $V = \mathsf{inr}\,(W)$

Case: $\triangleright\,(\alpha:\kappa).(\star\ \preceq_1^{\kappa,\mathsf{Val}}\ \langle\rangle)$ with $v\,[\alpha] = \mathbf{inl}\,(\star)$ and $V = \mathsf{inl}\,(\langle\rangle)$
It follows that $e_{\mathsf{case}}(v[\alpha]) = \Delta^\kappa([\![\mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle))]\!]^{\mathsf{Val},\kappa})$ and thus it remains to be shown that $\mathsf{step}^\kappa(\lambda\,(\alpha:\kappa).\Delta^\kappa([\![\mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle))]\!]^{\mathsf{Val},\kappa}))\ \preceq_\tau^{\kappa,\mathsf{Val}}\ \mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle))$, which by Lemma 2.4.2 is equivalent to

$$
(\triangleright^\kappa)^2\left(([\![\mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle))]\!]^{\mathsf{Val},\kappa})\ \preceq_\tau^{\kappa,\mathsf{Val}}\ \mathsf{fold}\,(\mathsf{inl}\,(\langle\rangle))\right).
$$

but this is implied by the fundamental lemma 2.5.15.

Case: $\triangleright\,(\alpha:\kappa).(w\ \preceq_{\tau\to\tau}^{\kappa,\mathsf{Val}}\ W)$ with $v\,[\alpha] = \mathbf{inr}\,(w)$ and $V = \mathsf{inr}\,(W)$
We have that $e_{\mathsf{case}}(v\,[\alpha]) = \Delta^\kappa([\![\mathsf{fold}\,(\mathsf{inr}\,(\mathsf{lam}\ y.h_{val}(w(h_{val}(y)))))]\!]^{\mathsf{Val},\kappa})$. It now follows that

$$
\begin{aligned}
&(\triangleright^\kappa)\left([\![\mathsf{lam}\ y.h_{val}(w(h_{val}(y)))]\!]^{\mathsf{Val},\kappa}\ \preceq_{\tau\to\tau}^{\kappa,\mathsf{Val}}\ W\right) && (2.2)\\
\to\ &(\triangleright^\kappa)^3\left([\![\mathsf{lam}\ y.h_{val}(w(h_{val}(y)))]\!]^{\mathsf{Val},\kappa}\ \preceq_{\tau\to\tau}^{\kappa,\mathsf{Val}}\ W\right) && (2.3)\\
\simeq\ &(\Delta^\kappa)^2([\![\mathsf{fold}\,(\mathsf{inr}\,(\mathsf{lam}\ y.h_{val}(w(h_{val}(y)))))]\!]^\kappa)\ \preceq_\tau^{\kappa,\mathsf{Tm}}\ \mathsf{fold}\,V && (2.4)\\
\simeq\ &(\triangleright^\kappa)^2\left(\mathsf{next}^\kappa(\mathbf{inr}\,([\![\mathsf{lam}\ y.h_{val}(w(h_{val}(y)))]\!]^{\mathsf{Val},\kappa}))\ \preceq_\tau^{\kappa,\mathsf{Val}}\ \mathsf{fold}\,(\mathsf{inr}\,(W))\right) && (2.5)\\
\simeq\ &\mathsf{step}^\kappa\,(\lambda\,(\alpha:\kappa).e_{\mathsf{case}}(v\,[\alpha]))\ \preceq_\tau^{\kappa,\mathsf{Tm}}\ \mathsf{fold}\,V && (2.6)
\end{aligned}
$$

and therefore, it suffices to prove Eq. (2.2). Thus, by definition of $\preceq^{\kappa,\mathsf{Val}}_{\tau\to\tau}$, it remains to show that

$$\triangleright^\kappa\left(\forall v', V'.(v' \preceq^{\kappa,\mathsf{Val}}_{\tau} V') \to [\![\mathsf{lam}\ y.h_{val}(w(h_{val}(y)))]\!]^{\mathsf{Val},\kappa}(v') \preceq^{\kappa,\mathsf{Tm}}_{\tau\to\tau} W(V)\right),$$

which is furthermore equivalent to

$$\triangleright^\kappa\left(\forall v', V'.(v' \preceq^{\kappa,\mathsf{Val}}_{\tau} V') \to ([\![h_{val}]\!]^\kappa \cdot ([\![w]\!]^\kappa \cdot ([\![h_{val}]\!]^\kappa \cdot [\![v']\!]^\kappa))) \preceq^{\kappa,\mathsf{Tm}}_{\tau\to\tau} W(V)\right).$$

This follows — just as for the previous direction — by applying Lemma 2.5.11 three times, while using the guarded hypothesis under tick.

2.7 Related works

In this paper, we have shown how to use synthetic guarded domain theory (SGDT) to model FPC. SGDT has been used in earlier work to model PCF [91], a call-by-name variant of FPC [82], FPC with general references [102, 103], untyped lambda calculus with nondeterminism [84], and guarded interaction trees [52]. Thus, the contribution of this technical report is a denotational semantics for call-by-value FPC together with an operationally sound relation in a guarded type theory.

Synthetic guarded domain theory [23] is an abstraction of metric domain theory [10]. These metric models have been used to model PCF by Escardó [50] already before the development of SGDT. Synthetic guarded domain theory has been extensively used to internalize programming languages and reason about their operational behavior. [23, 69]

There are different approaches to partiality in type theory. We use Capretta's [29] coinductive delay monad as a constructive way to represent potentially not terminating computations. It is well known that the resulting semantics is too intensional and needs to be quotiented by weak bisimilarity to have the correct equality. This turns out to be subtle [9, 34]. We instead define a logical relation that implies contextual refinement.

Our approach is inspired by Møgelberg and Paviotti [82], who followed a similar approach and defined a denotational semantics for call-by-name FPC, they used an extensional guarded type theory with clock quantification. Furthermore, Møgelberg and Vezzosi [88] used this technique to prove applicative may-similarity for an untyped lambda calculus with non-determinism.

There have been other synthetic approaches to domain theory: in particular synthetic domain theory — an abstraction of the classical Scott domain theory.

For instance, Rosolini and Hyland [64, 98] developed categorical models in which domains are certain kinds of sets. Fiore [51] axiomatized these models, and Reus [97] presented synthetic domain theory formally in the extended calculus of constructions together with extra axioms. Later on, Simpson [100] defined an interpretation of FPC using intuitionistic ZF set theory.

There are also analytic approaches to domain theory in type theory — that is, domain theory is developed inside a meta-theory, such as the Calculus of Constructions.

Examples of this are Benton et al. [18, 19], Dockins [46]. De Jong et al. [45] constructed domain theory in a predicative homotopy type theory and interpreted PCF in it.

Altenkirch et al. [9], Chapman et al. [34] used homotopy type theory and (in the case of [9]) quotient inductive inductive types to show that the Capretta's delay monad [29] quotiented by weak bisimilarity forms again a monad.

We work in Clocked Cubical Type Theory [72], since it is a well-known and developed guarded type theory, which — through the clock quantification type former — also has definable coinductive types. We anticipate that set quotient types are going to be handy in defining monads for other effects. (See [85, 88]). Besides that, this work could be carried out in other suitable modal type theories which provide a similar interface, such as [58].

2.8 Conclusion

We defined operational and denotational semantics for FPC in guarded type theory and constructed a relation between denotations and syntax to reason about contextual refinement. While this is the first such account in constructive guarded type theories, this result is not new and has been proven in different iterations before. As we see it, there are two considerations which justify the effort:

- This semantics is compositional and we conjecture that large parts of this work can be reused in more general settings with other effects.
- Constructive meta-theories allow proof assistant formalizations which makes complicated constructions more tractable.

In future work, we want to combine and extend these results to programming languages with other effects, such as probabilistic choice or non determinism.

Chapter 3

Modelling Probabilistic FPC in Guarded Type Theory

Abstract

Type theory combines logic and programming in one language. This is useful both for reasoning about programs written in type theory, as well as for reasoning about other programming languages inside type theory. It is well-known that it is challenging to extend these applications to languages with recursion and computational effects such as probabilistic choice, because these features are not easily represented in type theory.

We show how to define and reason about $\mathsf{FPC}_{\oplus}$, a programming language with probabilistic choice and recursive types, in guarded type theory. We use higher inductive types to represent finite distributions and guarded recursion to model recursion. We define both operational and denotational semantics of $\mathsf{FPC}_{\oplus}$ and prove soundness. Finally, we construct a relation between the two and show how to use this to reason about programs up to contextual equivalence.

3.1 Introduction

Probabilistic programming languages include commands that generate random values by sampling from a probability distribution. Thus, a probabilistic program evaluates to a distribution of values, as opposed to a single value in the case of ordinary deterministic computation. It is well known that it is challenging to develop semantic models for reasoning about higher-order probabilistic programming languages including recursion, even in a classical meta-theory. Nonetheless, a plethora of denotational approaches have been investigated in recent years, [43, 48, 61, 68, 107]. Other operational-based approaches to reason about probabilistic programs have also been shown to scale to rich languages with a variety of features, using techniques such as logical relations [26, 42, 67, 111, 113], or bisimulations [41, 73].

In this paper, we investigate how one can develop operational and denotational semantics of $\mathsf{FPC}_{\oplus}$, a call-by-value higher-order probabilistic programming language

with recursive types, in (a variant of) *constructive type theory*, and prove that the denotational semantics is adequate with respect to the operational semantics, paving the way to formalisations of programming language semantics in type theory. There are several technical challenges to meet this end: 1. The standard (classical) operational semantics of $\mathsf{FPC}_{\oplus}$ relies on real numbers and on classical reasoning, not available in constructive type theory. 2. $\mathsf{FPC}_{\oplus}$ includes non-terminating computations, which means that our denotational semantics must use some kind of domain theory. 3. $\mathsf{FPC}_{\oplus}$ includes recursive types, which means that we must be able to define domains recursively in some way.

To address the latter two challenges, we will make use of a form of *guarded* type theory, which can be thought of as an abstract approach to step-indexing techniques [11, 12] and metric domain theory [10, 49]. Indeed, in guarded type theory one can use guarded recursive types to develop *synthetic guarded domain theory* [23], which has been used in earlier work to model (ordinary, non-probabilistic, call-by-name) FPC [82].

The specific guarded type theory we use in this paper is Clocked Cubical Type Theory (CCTT). It includes a modal type-operator $\triangleright^{\kappa}$, indexed by a so-called clock κ (see Section 3.2), to describe data that is available one time step from now, and it is possible to define elements by guarded recursion by means of a fixed point combinator $\mathsf{fix}^{\kappa} : (\triangleright^{\kappa} A \to A) \to A$. By applying the fixed point combinator to an operator on a universe one can obtain solutions to guarded recursive type equations [21]. For example, one can define a *guarded delay monad* L^{κ}, which maps a type A to $L^{\kappa}A$ satisfying $L^{\kappa}A \simeq A + \triangleright^{\kappa}(L^{\kappa}A)$. This guarded delay monad is used in [82, 91] to model non-terminating computations and in the guarded domain theory of *op. cit.*, a guarded domain is then an algebra for this monad.

To address the first challenge, we define a novel combined distribution (for probabilistic choice) and delay (for non-termination) monad in synthetic guarded domain theory, which we refer to as a *guarded convex delay monad* and which we denote by D^{κ}. On a type A, $\mathsf{D}^{\kappa}A \simeq \mathscr{D}(A + \triangleright^{\kappa}(\mathsf{D}^{\kappa}A))$, where $\mathscr{D}$ is a finite distribution monad. Intuitively, this means that a computation of type A will be a distribution over values of A (immediately available) and delayed computations of type A.

Clocked Cubical Type Theory not only models guarded recursion but also higher-inductive types (HITs), which we use to define the finite distribution monad $\mathscr{D}$: on a set A, $\mathscr{D}A$ is the free convex algebra [66], that is, a set $\mathscr{D}A$ together with a binary operation $\oplus_p$, indexed by a rational number p, satisfying the natural equational theory (idempotency, associativity, and commutativity). Using a HIT to represent distributions, as opposed to a partial function from A to the rationals between 0 and 1, allows us to use the equational properties (associativity and commutativity) of the meta-language level when reasoning about the denotational semantics of $\mathsf{FPC}_{\oplus}$, and it provides us with a useful induction principle for proving propositions ranging over $\mathscr{D}A$.

We define a notion of contextual refinement for $\mathsf{FPC}_{\oplus}$ terms, by closing terms under contexts of unit type and comparing their probabilities of termination, as is standard. In contrast to the classical setting, we only use finite approximations since

the limits of termination probabilities cannot be computed in finite time. Moreover, we define a logical relation, which relates the denotational and operational semantics, and prove that it is sound with respect to contextual refinement. Traditionally, defining a logical relation relating denotational and operational semantics for a language with recursive types is non-trivial, see, e.g., [96]. Here we use the guarded type theory to define the logical relation by guarded recursion. As usual, the logical relation is divided into a relation for values and a relation for computations. Since computations compute to distributions, the challenge here lies in defining the relation for computations in terms of the relation for values. Earlier work on operationally-based logical relations [7, 26] used bi-orthogonality to reduce the problem to relating termination probabilities for computations of ground type. Here, we follow the approach of the recent article [59], which uses couplings [16, 78, 105, 110] to lift relations on values to relations on distributions. Note that the development in [59] relies on classical logic (in particular, the composition of couplings, the so-called bind lemma, relies on the axiom of choice) and thus does not apply here. Instead, we define a novel constructive notion of lifting of relations for convex delay algebras, for which we establish a series of basic results, including a version of the important bind lemma that allows us to compose these liftings in proofs.

Finally, we use the semantics and the logical relation to prove contextual refinement of examples that combine probabilistic choice and recursion.

Contributions

1. To the best of our knowledge, we present the first constructive type theoretic account of operational and denotational semantics of $\mathsf{FPC}_{\oplus}$.
2. Our denotational semantics makes use of a novel guarded convex delay monad, which is defined as the solution to a guarded recursive type equation, and which can be understood as a natural generalization of earlier guarded delay monads.
3. We develop the basic constructive theory of couplings for convex delay algebras and use it to define a logical relation, relating denotational and operational semantics.
4. We demonstrate how to use the semantics to reason about examples that combine probabilistic choice and recursion.

3.2 Clocked Cubical Type Theory

We will work in Clocked Cubical Type Theory (CCTT) [72], an extension of Cubical Type Theory [37] with guarded recursion and higher inductive types. At present, CCTT is the only existing type theory containing all constructions needed for this paper. We will not describe CCTT in detail, but only describe the properties we need, with the hope of making the paper more accessible, and so that the results can be reused in other (future) type theories with the same properties.

Basic properties and HITs

CCTT has an infinite hierarchy of (Tarski style) universes (U_i), as well as identity types satisfying the standard rules and function extensionality. Precisely, CCTT, being based on Cubical Type Theory, has a path type as primitive, rather than an identity type in the traditional sense. However, using the path type, one can encode an equivalent identity type [37]. We will write $a =_A b$, or just $a = b$, for the identity type associated with terms $a, b : A$ of the same type. Following conventions from homotopy type theory [106], we say that a type A is a (homotopy) proposition if for any $x, y : A$, the type $x = y$ is contractible, and that it is a (homotopy) set if for any $x, y : A$, the type $x = y$ is a proposition. We write $\mathsf{Prop}[i]$, and Set_i for the subuniverses of U_i of propositions and sets respectively, often omitting the universe level i. We write $A \simeq B$ for the type of equivalences from A to B. We shall mostly use this in the case where A and B are sets, and in that case, an equivalence is simply given by the standard notion of isomorphism of sets as phrased inside type theory using propositional equality.

CCTT also has higher inductive types (HITs), and we will use these to construct propositional truncation, set truncation and the finite distributions monad (see section 3.3). In particular, this means that one can express ordinary propositional logic with operators $\wedge, \vee, \exists, \forall$ on Prop, using the encodings of $\exists$ and $\vee$ defined using propositional truncation [106]. Recall in particular the elimination principles for $\exists$: When proving a proposition ψ assuming $\exists(x : X).\phi(x)$ we may assume we have an x in hand satisfying $\phi(x)$, but we cannot do that when mapping from $\exists(x : X).\phi(x)$ to an arbitrary type. We will also need inductive types to represent the type $\mathcal{N}$ of natural numbers, as well as types and terms of the language $\mathsf{FPC}_{\oplus}$ described in section 3.5. These are all captured by the schema for higher inductive families in CCTT [72].

Guarded recursion

Guarded recursion uses a modal operator $\triangleright$ (pronounced ´later') to describe data that is available one time step from now. In multiclocked guarded recursion, $\triangleright$ is indexed by a clock κ. Clocks can be variables of the pretype clock or they can be the clock constant κ_0. Clocks can be universally quantified in the type $\forall\kappa.A$, with rules similar to Π-types, including a functional extensionality principle. Introduction and elimination for $\triangleright$ is by abstraction and application to *ticks*, i.e., assumptions of the form $\alpha : \kappa$, to be thought of as evidence that time has ticked on clock κ. Because ticks can occur in terms, they can also occur in types, and the type $\triangleright(\alpha : \kappa).A$ binds α in A. When α does not occur in A we simply write $\triangleright^{\kappa} A$ for $\triangleright(\alpha : \kappa).A$. The rules for tick abstraction and application that we shall use in this paper are presented in Figure 3.1. Note that the rule for tick application assumes that β (or anything occurring after that in the context) does not already occur in t. This is to avoid terms of the type $\triangleright^{\kappa}\triangleright^{\kappa} A \to \triangleright^{\kappa} A$ merging two time steps into one. We will sometimes use the notation

$$\mathsf{next}^{\kappa} \triangleq \lambda x.\lambda(\alpha : \kappa).x : A \to \triangleright^{\kappa} A. \tag{3.1}$$

The rules presented in Figure 3.1 are special cases of those of CCTT. The general rules allow certain 'timeless' assumptions in Γ' to occur in t in the tick application

$$\frac{\kappa : \mathsf{clock} \in \Gamma}{\Gamma, \alpha : \kappa \vdash} \qquad \frac{\Gamma \vdash t : \triangleright(\alpha : \kappa).A \qquad \Gamma, \beta : \kappa, \Gamma' \vdash}{\Gamma, \beta : \kappa, \Gamma' \vdash t\,[\beta] : A\,[\beta/\alpha]}$$

$$\frac{\Gamma, \alpha : \kappa \vdash t : A}{\Gamma \vdash \lambda(\alpha : \kappa).t : \triangleright(\alpha : \kappa).A} \qquad \frac{\Gamma, \kappa : \mathsf{clock} \vdash t : A}{\Gamma \vdash \Lambda\kappa.t : \forall\kappa.A}$$

$$\frac{\Gamma \vdash t : \forall\kappa.A \qquad \Gamma \vdash \kappa' : \mathsf{clock}}{\Gamma \vdash t[\kappa'] : A[\kappa'/\kappa]} \qquad \frac{\Gamma \vdash t : \triangleright^\kappa A \to A}{\Gamma \vdash \mathsf{dfix}^\kappa t : \triangleright^\kappa A}$$

$$\frac{\Gamma \vdash t : \triangleright^\kappa A \to A}{\Gamma \vdash \mathsf{pfix}^\kappa t : \triangleright(\alpha : \kappa).(\mathsf{dfix}^\kappa t)\,[\alpha] =_A t(\mathsf{dfix}^\kappa t)}$$

Figure 3.1: Selected typing rules for Clocked Cubical Type Theory.

rule. This allows typing of an extensionality principle for $\triangleright$ of type

$$(x =_{\triangleright^\kappa A} y) \simeq \triangleright(\alpha : \kappa).(x\,[\alpha] =_A y\,[\alpha]). \tag{3.2}$$

In this paper we shall simply take this as an axiom. One consequence of (3.2) is that $\triangleright$ preserves the property of being a set or a proposition. Other omitted rules for ticks allow typing of a *tick-irrelevance* axiom

$$\mathsf{tirr}^\kappa : \Pi(x : \triangleright^\kappa A).\triangleright(\alpha : \kappa).\triangleright(\beta : \kappa).(x\,[\alpha]) =_A (x\,[\beta]). \tag{3.3}$$

The fixed point combinator dfix allows for encoding and programming with guarded recursive types. Define $\mathsf{fix}^\kappa : (\triangleright^\kappa A \to A) \to A$ as $\mathsf{fix}^\kappa f = f(\mathsf{dfix}^\kappa f)$. Then one can prove that

$$\mathsf{fix}^\kappa t = t(\lambda(\alpha : \kappa).\mathsf{fix}^\kappa t). \tag{3.4}$$

Applying the fix point operator to maps on a universe, as in [22], one can encode *guarded recursive types* such as the *guarded delay monad* L^κ mapping a type A to $L^\kappa A$ satisfying

$$L^\kappa A \simeq A + \triangleright^\kappa(L^\kappa A). \tag{3.5}$$

In this paper, we will not spell out how such guarded recursive types are defined as fixed points, but just give the defining guarded recursive equation. Intuitively, the reason this is well defined is that $L^\kappa A$ only occurs on the right-hand side of (3.5) under a $\triangleright$, which allows the recursive equation to be phrased as a map $\triangleright^\kappa \mathsf{U} \to \mathsf{U}$. One can also use fix to program with L^κ defining, e.g., the diverging computation as $\bot = \mathsf{fix}(\lambda x.\mathsf{inr}(x))$ (leaving the type equivalence between $L^\kappa A$ and its unfolding implicit). In this case (3.4) specialises to $\bot = \mathsf{inr}(\lambda(\alpha : \kappa).\bot)$.

Coinductive types

Coinductive types can be encoded by quantifying over clocks in guarded recursive types [13]. For example, the type $LA \triangleq \forall\kappa.L^\kappa A$ defines a coinductive solution to $LA \simeq A + LA$ in CCTT, provided A is *clock-irrelevant*, meaning that the canonical map $A \to \forall\kappa.A$ is an equivalence. More generally, one can prove that any functor $F : \mathsf{U} \to \mathsf{U}$ (in the naive sense of having a functorial action $(A \to B) \to FA \to FB$) commuting with clock quantification in the sense that $F(\forall\kappa.X) \simeq \forall\kappa.F(X)$ via the canonical map, has a final coalgebra defined as $\nu(F) \triangleq \forall\kappa.\nu^\kappa(F)$, where $\nu^\kappa(F) \simeq F(\triangleright^\kappa(\nu^\kappa(F))$ is defined using fix. This encoding also works for indexed coinductive types. The correctness of the encoding of coinductive types can be proved in CCTT, and relies on the type equivalence

$$\forall\kappa.A \simeq \forall\kappa.\triangleright^\kappa A.$$

For the encoding to be useful, one needs a large collection of clock-irrelevant types and functors commuting with clock quantification. We will need that $\mathscr{N}$ is clock-irrelevant, and that clock quantification commutes with sums in the sense that $\forall\kappa.(A+B) \simeq (\forall\kappa.A) + (\forall\kappa.B)$. Both of these can be proved in CCTT using the notion of induction under clocks for higher inductive types. Note also that all propositions P are clock-irrelevant, because the clock constant κ_0 can be used to define a map $(\forall\kappa.P) \to P$. For any other types that we need to be clock-irrelevant, we use the following.

Lemma 3.2.1. *Suppose $i : A \to B$ is injective in the sense of the existence of a map $\Pi x, y : A.(i(x) = i(y)) \to x = y$, and suppose B is clock irrelevant. Then also A is clock-irrelevant.*

3.3 Finite distributions

For the rest of the paper, we work informally in CCTT. We start by constructing a monad $\mathscr{D}$ for finite distributions on a set. We will assume a type representing the open rational interval $(0,1)$. In practice, there are several ways of specifying this, for example as a type of pairs (n,d), of mutually prime, positive natural numbers satisfying $n < d$. We will not specify how to do this, we just need a few operations: product, division, and inversion $(1-(-))$, as well as the fact that $(0,1)$ is a set with decidable equality, and that $(0,1)$ is clock irrelevant. The latter follows from the embedding into $\mathscr{N} \times \mathscr{N}$ and Lemma 3.2.1. Likewise, we need a representation of $[0,1]$, obtained, e.g., by just adding two points to $(0,1)$.

In classical presentations of probability theory, a finite distribution on a set A is a finite map into $[0,1]$, whose values sum to 1. In type theory, this would be represented by a subtype of $A \to [0,1]$, which would need some notion of finite support, as well as a definition of the functorial action of $\mathscr{D}$. It is unclear to us how to do this without assuming decidable equality. Another approach could be to use lists of key-value pairs, but this requires a quotient to obtain the correct notion of equality of distributions.

Here we instead choose to represent $\mathscr{D}$ as the free monad for the theory of convex algebras, which is known to generate the finite distribution monad [66].

Definition 3.3.1 (Convex Algebra)**.** A *convex algebra* is a set A together with an operation $\oplus : (0,1) \to A \to A \to A$, such that

$$\mu \oplus_p \mu = \mu \tag{idem}$$
$$\mu \oplus_p \nu = \nu \oplus_{1-p} \mu \tag{comm}$$
$$(\mu_1 \oplus_p \mu_2) \oplus_q \mu_3 = \mu_1 \oplus_{pq} \left(\mu_2 \oplus_{\frac{q-pq}{1-pq}} \mu_3\right) \tag{assoc}$$

A map $f : A \to B$ between convex algebras A and B is a *homomorphism* if $f(\mu \oplus_p \nu) = f(\mu) \oplus_p f(\nu)$ holds for all μ, ν, p.

Definition 3.3.2. Let $\mathscr{D}(A)$ be the higher inductive type defined using two constructors

$$\delta : A \to \mathscr{D}(A) \qquad \oplus : (0,1) \to \mathscr{D}(A) \to \mathscr{D}(A) \to \mathscr{D}(A)$$

and equations for idempotency, commutativity and associativity as in the definition of convex algebra, plus an equation for set truncation.

This definition can be formalised as a HIT in CCTT, similarly to the definition of the finite powerset [72]. With that definition, the recursion principle for the HIT exactly corresponds to $\mathscr{D}(A)$ being a free convex algebra.

Proposition 3.3.3. *$\mathscr{D}(A)$ is the free convex algebra on A, in the sense that for any convex algebra B, and function $f : A \to B$, there exists a unique homomorphism of convex algebras $\overline{f} : \mathscr{D}(A) \to B$ satisfying $f = \overline{f} \circ \delta$. As a consequence, $\mathscr{D}(-)$ forms a monad on the category of sets.*

Recall also the following induction principle for the HIT $\mathscr{D}(A)$: If $\phi(x)$ is a proposition for all $x : \mathscr{D}(A)$ and $\phi(\delta(a))$ holds for all a, and moreover, $\phi(\mu)$ and $\phi(\nu)$ implies $\phi(\mu \oplus_p \nu)$ for all μ, ν, p, then $\phi(x)$ holds for all x.

If X has decidable equality, one can associate a probability function $p_\mu : X \to [0,1]$ by induction on the distribution μ using $p_{\delta(x)}(y) = 1$ if $x = y$ and $p_{\delta(x)}(y) = 0$ else. Define the Bishop finite sets in the standard way by induction on n as $\mathsf{Fin}(0) = 0$ and $\mathsf{Fin}(n+1) = \mathsf{Fin}(n) + 1$. For these sets, we can relate $\mathscr{D}$ to its classical definition.

Lemma 3.3.4. *$\mathscr{D}(\mathsf{Fin}(n)) \simeq \Sigma(f : \mathsf{Fin}(n) \to [0,1]).\mathsf{sum}(f) = 1$, where sum is the sum of the values of f.*

Proof. Let $\mathscr{D}'(\mathsf{Fin}(n)) \triangleq \Sigma(f : \mathsf{Fin}(n) \to [0,1]).\mathsf{sum}(f) = 1$, and note that this carries a convex algebra structure as well as contains $\mathsf{Fin}(n)$ via a dirac map δ both defined in the standard way. We show that this is free by induction on n. In the case of $n = 0$, clearly $\mathscr{D}'(\mathsf{Fin}(0))$ is empty, and so the case follows. For the inductive case, let n be the element of $\mathsf{Fin}(n+1)$ not in the inclusion from $\mathsf{Fin}(n)$. Given $f : \mathscr{D}'(\mathsf{Fin}(n+1))$ we must define $\overline{g}(f)$. By decidability of equality for $[0,1]$, we can branch on the values

of $f(n)$. If $f(n) = 0$, then f is in the image of the obvious inclusion from $\mathscr{D}'(\mathsf{Fin}(n))$, and so $\overline{g}(f)$ is defined by induction. If $f(n) = 1$ then $f = \delta(n)$ and so $\overline{g}(f)$ must be $g(n)$. Finally, if $f(n) = p \in (0,1)$, then $f = \delta(n) \oplus_p f'$ for some $f' : \mathscr{D}'(\mathsf{Fin}(n))$. By induction we get $\overline{g}(f')$ and so can define $\overline{g}(f) = g(n) \oplus_p \overline{g}(f')$.

To prove that $\overline{g}$ is a homomorphism, suppose we are given $f : \mathscr{D}'(\mathsf{Fin}(n+1))$ and $h : \mathscr{D}'(\mathsf{Fin}(n+1))$. There are nine possible cases of $f(n)$ and $h(n)$ and we just show the case where $f(n) = q$ and $h(n) = r$. In that case $f = \delta(n) \oplus_q f'$ and $h = \delta(n) \oplus_r h'$ for some $f', h' : \mathscr{D}'(\mathsf{Fin}(n+1))$. From the axioms of convex algebras, one can compute, given $p, q, r : (0,1)$, probabilities p', q', r' such that $(a \oplus_q b) \oplus_p (c \oplus_r d) = (a \oplus_{q'} c) \oplus_{p'} (b \oplus_{r'} d)$ holds for all a, b, c, d in any convex algebra. We use this to get

$$\begin{aligned}
\overline{g}(f \oplus_p h) &= \overline{g}(\delta(n) \oplus_{p'} (f' \oplus_{r'} h')) \\
&= g(n) \oplus_{p'} \overline{g}(f' \oplus_{r'} h') \\
&= g(n) \oplus_{p'} (\overline{g}(f') \oplus_{r'} \overline{g}(h')) \\
&= (g(n) \oplus_{q'} g(n)) \oplus_{p'} (\overline{g}(f') \oplus_{r'} \overline{g}(h')) \\
&= (g(n) \oplus_q \overline{g}(f')) \oplus_p (g(n) \oplus_r \overline{g}(h')) \\
&= \overline{g}(f) \oplus_p \overline{g}(h).
\end{aligned}$$

Using the induction step to conclude $\overline{g}(f' \oplus_{r'} h') = \overline{g}(f') \oplus_{r'} \overline{g}(h')$ in the third equality. □

Example 23. The isomorphism of Lemma 3.3.4 allows one to prove equations of distributions by mapping to the right-hand side and using functional extensionality. Consider, for example, the equation

$$(a \oplus_p b) \oplus_p (c \oplus_p a) = (b \oplus_{\frac{1}{2}} c) \oplus_{2p(1-p)} a,$$

where $a, b, c : \mathscr{D}X$ for some X. In the case where $X = \mathsf{Fin}(3)$ and $a = \delta(0), b = \delta(1), c = \delta(2)$, we can prove this by noting that both the left and right-hand sides of the equation correspond to the map $f(0) = p^2 + (1-p)^2 = 1 + 2p^2 - 2p = 1 - 2p(1-p)$, $f(1) = f(2) = p(1-p)$. Finally, the case of general X, a, b, c follows from applying functoriality to the canonical map $\mathsf{Fin}(3) \to \mathscr{D}X$.

In constructive mathematics, a set is said to be Kuratowski-finite if it is the codomain of a bijection from a set of the form $\mathsf{Fin}(n)$. The distributions of $\mathscr{D}(A)$ satisfy a similar property.

Lemma 3.3.5. *For any $\mu : \mathscr{D}(A)$, there exists an $n : \mathscr{N}$, a map $f : \mathsf{Fin}(n) \to A$, and a distribution $\nu : \mathscr{D}(\mathsf{Fin}(n))$ such that $\mu = \mathscr{D}(f)(\nu)$*

Proof. This is proved by induction on μ. In the case of $\delta(a)$ take $n = 1$ and f the constant map to a. In case of $\mu \oplus_p \mu'$ we get by induction $n, n', f : \mathsf{Fin}(n) \to A, f' : \mathsf{Fin}(n') \to A, \nu : \mathscr{D}(\mathsf{Fin}(n)), \nu' : \mathscr{D}(\mathsf{Fin}(n'))$ such that $\mu = \mathscr{D}(f)(\nu)$ and $\mu' = \mathscr{D}(f)(\nu')$. Set $m = n + n'$ such that $\mathsf{Fin}(m) \simeq \mathsf{Fin}(n) + \mathsf{Fin}(n')$, define $\rho \triangleq \mathscr{D}(\mathsf{inl})(\nu)$, $\rho' \triangleq \mathscr{D}(\mathsf{inr})(\nu')$ and $g : \mathsf{Fin}(m) \to A$ to be the copairing of f and f'. Then $\mathscr{D}(g)(\rho \oplus_p \rho') = \mu \oplus_p \mu'$. □

Distributions on sum types

The aim of this subsection is to prove that any distribution $\mu : \mathscr{D}(A+B)$ on a sum type can be written uniquely as a convex sum of two subdistributions: One on A and one on B. In the classical setting where a distribution is a map with finite support this is trivial: The two subdistributions are simply the normalised restrictions of the distribution map to A and B respectively. In the type theoretic setting, the proof requires a bit more work. We start by considering the special case where B is the unit type.

Lemma 3.3.6. *The types $\mathscr{D}(A+1)$ and $1+[0,1)\times\mathscr{D}A$ are equivalent.*

Proof (sketch). Using the classical definition of $\mathscr{D}$ in terms of finite maps, the equivalence $\mathscr{D}(A+1)\to 1+[0,1)\times\mathscr{D}A$ would map f to $\star$ if $f(\star)=1$ and otherwise to the pair $(f(\star), \frac{1}{1-f(\star)}\cdot g)$ where g is the restriction of f to A. This isomorphism transports the convex algebra structure on $\mathscr{D}(A+1)$ to one on $1+[0,1)\times\mathscr{D}A$ defined by the following clauses.

$$\begin{aligned}
\star \oplus_p \star &\triangleq \star \\
(q,\mu)\oplus_p \star &\triangleq (pq+(1-p),\mu) \\
\star \oplus_p (r,\nu) &\triangleq (p+(1-p)r,\nu) \\
(q,\mu)\oplus_p (r,\nu) &\triangleq (pq+(1-p)r, \mu \oplus_{\left(\frac{p(1-q)}{p(1-q)+(1-p)(1-r)}\right)} \nu)
\end{aligned}$$

To prove the lemma in type theory, it suffices to prove that the above equations define a convex algebra structure on $1+[0,1)\times\mathscr{D}A$ and that this is the free convex algebra on $A+1$. Proving that the axioms of convex algebras are satisfied is quite tedious, but there is another approach: First note that the classical proof actually works in the case of $A=\mathsf{Fin}(n)$ using $Fin(n)+1=\mathsf{Fin}(n+1)$ and Lemma 3.3.4, then apply Lemma 3.3.5 to show that any equation in $1+[0,1)\times\mathscr{D}A$ is the image of one in $1+[0,1)\times\mathscr{D}(\mathsf{Fin}(n))$ under some map $\mathsf{Fin}(n)\to A$. Since the equations hold in $1+[0,1)\times\mathscr{D}(\mathsf{Fin}(n))$ they also hold in $1+[0,1)\times\mathscr{D}(A)$. □

We can now prove the more general theorem.

Theorem 3.3.7. *For all sets A and B, the map*

$$\mathscr{D}(A)+\mathscr{D}(B)+\mathscr{D}(A)\times(0,1)\times\mathscr{D}(B)\to\mathscr{D}(A+B),$$

defined by

$$\begin{aligned}
f(\mathsf{in}_1(\mu)) &\triangleq \mathscr{D}(\mathsf{inl})(\mu) \\
f(\mathsf{in}_2(\mu)) &\triangleq \mathscr{D}(\mathsf{inr})(\mu) \\
f(\mathsf{in}_3(\mu,p,\nu)) &\triangleq (\mathscr{D}(\mathsf{inl})(\mu))\oplus_p(\mathscr{D}(\mathsf{inr})(\nu)),
\end{aligned}$$

is an equivalence of types.

Proof (sketch). Since the domain and codomain are both sets, it suffices to prove that f is both surjective and injective.

To prove surjectivity, suppose $\mu : \mathscr{D}(A+B)$. We show that μ is in the image of f by induction on μ. The case of μ being a Dirac distribution is easy. In the case of a sum $\mu \oplus_p \nu$, one must consider the 9 cases of the induction hypotheses for μ and ν. Here we just do the case where $\mu = \mu_A \oplus_q \mu_B$ and $\nu = \nu_A \oplus_q \nu_B$, omitting the $\mathscr{D}(\mathsf{inl})$ and $\mathscr{D}(\mathsf{inr})$. Let p', q', r' be the unique elements in $(0,1)$ such that

$$(a \oplus_q b) \oplus_p (c \oplus_r d) = (a \oplus_{q'} c) \oplus_{p'} (b \oplus_{r'} d)$$

holds for all a, b, c, d. Then

$$\begin{aligned}(\mu_A \oplus_{q'} \nu_A) \oplus_{p'} (\mu_B \oplus_{r'} \nu_B) &= (\mu_A \oplus_q \mu_B) \oplus_p (\nu_A \oplus_r \nu_B) \\ &= \mu \oplus_p \nu.\end{aligned}$$

To prove injectivity, there are 8 cases to cover. Suppose for example that $\mu_A \oplus_p \mu_B = \nu_A \oplus_p \nu_B$ for $\mu_A, \nu_A : \mathscr{D}(A)$ and $\mu_B, \nu_B : \mathscr{D}(B)$. Then also

$$\begin{aligned}\mu_A \oplus_p \delta(\star) &= \mathscr{D}(A+!)(\mu_A \oplus_p \mu_B) \\ &= \mathscr{D}(A+!)(\nu_A \oplus_q \nu_B) \\ &= \nu_A \oplus_q \delta(\star),\end{aligned}$$

which via the equivalence of Lemma 3.3.6 allows us to conclude that $p = q$ and $\mu_A = \nu_A$. Similarly, we can prove that $\mu_B = \nu_B$. The other seven cases are similar. □

Alternatively, one can prove Theorem 3.3.7 directly by constructing a convex algebra structure on the 3-fold sum. Proving Lemma 3.3.6 first reduces the number of cases for associativity from 27 to 8.

3.4 Convex delay algebras

In this section, we define the guarded convex delay monad D^κ and the convex delay monad $\mathsf{D}^\forall$ modelling the combination of probabilistic choice and recursion. We first recall the notion of delay algebra (sometimes called a lifting or a $\triangleright^\kappa$-algebra [91]) and define a notion of convex delay algebra.

Definition 3.4.1 (Convex Delay Algebra)**.** A (κ-)*delay algebra* is a set A together with a map $\mathsf{step}^\kappa : \triangleright^\kappa A \to A$. A delay algebra homomorphism is a map $f : A \to B$ such that $f(\mathsf{step}^\kappa(a)) = \mathsf{step}^\kappa(\lambda(\alpha:\kappa).f(a[\alpha]))$ for all $a : \triangleright^\kappa A$. A *convex delay algebra* is a set A with both a delay algebra structure and a convex algebra structure, and a homomorphism of these is a map respecting both structures.

The type $L^\kappa A$ is easily seen to be the free guarded delay algebra on a set A. Define the *guarded convex delay monad* as the guarded recursive type

$$\mathsf{D}^\kappa A \simeq \mathscr{D}(A + \triangleright^\kappa(\mathsf{D}^\kappa A))$$

Again, this can be constructed formally as a fixed point on a universe (assuming A lives in the same universe), but we shall not spell this out here. To see that this is a monad, we show that it is a free convex delay algebra. First define the convex delay algebra structure $(\mathsf{step}^\kappa, \delta^\kappa, \oplus^\kappa)$ on $\mathsf{D}^\kappa A$ as

$$\mathsf{step}^\kappa(a) = \delta(\mathsf{inr}(a)) \qquad \delta^\kappa a = \delta(\mathsf{inl}(a)) \qquad \mu \oplus_p^\kappa \nu = \mu \oplus_p \nu$$

where the convex algebra structure on the right-hand sides of the equations above refers to those of $\mathscr{D}$.

Proposition 3.4.2. *$\mathsf{D}^\kappa A$ is the free convex delay algebra structure on A, for any set A.*

The proof uses standard guarded recursion techniques. We include it in the paper to illustrate this technique for readers less familiar with guarded recursion.

Proof. Suppose $f : A \to B$ and that B is a convex delay algebra. We define the extension $\overline{f} : \mathsf{D}^\kappa A \to B$ by guarded recursion, so suppose we are given $g : \triangleright^\kappa(\mathsf{D}^\kappa A \to B)$ and define

$$\begin{aligned}
\overline{f}(\mathsf{step}^\kappa(a)) &= \mathsf{step}^\kappa(\lambda(\alpha:\kappa).g\,[\alpha](a\,[\alpha])) \\
\overline{f}(\delta^\kappa(a)) &= f(a) \\
\overline{f}(\mu \oplus_p^\kappa \nu) &= \overline{f}(\mu) \oplus_p \overline{f}(\nu)
\end{aligned}$$

Note that these cases define $\overline{f}$ by induction on $\mathscr{D}$ and $+$. Unfolding the guarded recursive definition using (3.4) gives

$$\overline{f}(\mathsf{step}^\kappa(a)) = \mathsf{step}^\kappa(\lambda(\alpha:\kappa).\overline{f}(a\,[\alpha]))$$

so that $\overline{f}$ is a homomorphism of convex delay algebras. For uniqueness, suppose g is another homomorphism extending f. We show that $g = \overline{f}$ by guarded recursion and function extensionality. So suppose we are given $p : \triangleright^\kappa(\Pi(x : \mathsf{D}^\kappa A)(g(a) = \overline{f}(a)))$. Then, for any $a : \triangleright^\kappa(\mathsf{D}^\kappa A)$, the term $\lambda(\alpha:\kappa).p\,[\alpha](a\,[\alpha])$ proves

$$\triangleright(\alpha:\kappa).(g(a\,[\alpha]) = \overline{f}(a\,[\alpha]))$$

which by (3.2) is equivalent to

$$\lambda(\alpha:\kappa).g(a\,[\alpha]) = \lambda(\alpha:\kappa).\overline{f}(a\,[\alpha])$$

Then,

$$\begin{aligned}
g(\mathsf{step}^\kappa(a)) &= \mathsf{step}^\kappa(\lambda(\alpha:\kappa).g(a\,[\alpha])) \\
&= \mathsf{step}^\kappa(\lambda(\alpha:\kappa).\overline{f}(a\,[\alpha])) \\
&= \overline{f}(\mathsf{step}^\kappa(a))
\end{aligned}$$

The rest of the proof that $g(a) = \overline{f}(a)$, for all a, then follows by HIT induction on $\mathscr{D}$. □

We will often write $t >>=^{\kappa} f$ for $\overline{f}(t)$ where $\overline{f}$ is the unique extension of f to a convex delay algebra homomorphism.

Example 24. The type $\mathsf{D}^{\kappa}(X)$ can be thought of as a type of probabilistic processes returning values in X. Define, for example, a geometric process with probability $p : (0,1)$ as $\mathsf{geo}_p^{\kappa} 0$ where

$$\mathsf{geo}_p^{\kappa} : \mathscr{N} \to \mathsf{D}^{\kappa}(\mathscr{N})$$
$$\mathsf{geo}_p^{\kappa} n \triangleq (\delta^{\kappa} n) \oplus_p \mathsf{step}^{\kappa}(\lambda(\alpha : \kappa).\mathsf{geo}_p^{\kappa}(n+1))$$

Note that this gives

$$\begin{aligned}\mathsf{geo}_p^{\kappa} &= (\delta^{\kappa} 0) \oplus_p \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\mathsf{geo}_p^{\kappa}(1)) \\ &= (\delta^{\kappa} 0) \oplus_p \left(\mathsf{step}^{\kappa}\lambda(\alpha:\kappa).\left((\delta^{\kappa} 1) \oplus_p (\mathsf{step}^{\kappa}\lambda(\alpha:\kappa).\mathsf{geo}_p^{\kappa}(2))\right)\right) \\ &= \ldots\end{aligned}$$

The modal delay $\triangleright^{\kappa}$ in the definition of $\mathsf{D}^{\kappa}(X)$ prevents us from accessing values computed later, and, *e.g.*, compute probabilities of termination in n steps as elements of $[0,1]$. Such operations should instead be defined on the convex delay monad $\mathsf{D}^{\forall}$ defined as

$$\mathsf{D}^{\forall} A \triangleq \forall\kappa.\mathsf{D}^{\kappa} A$$

Proposition 3.4.3. *If A is clock-irrelevant then $\mathsf{D}^{\forall} A$ is the final coalgebra for the functor $F(X) = \mathscr{D}(A + X)$.*

Proof. We must show that F commutes with clock quantification, and this reduces easily to showing that $\mathscr{D}$ commutes with clock quantification. The latter can be either proved directly by using the same technique as for the similar result for the finite powerset functor [72], or by referring to [85, Proposition 14], which states that any free model monad for a theory with finite arities commutes with clock quantification. □

Remark 25. From now on, whenever we look at a type $\mathsf{D}^{\forall} A$, we assume A to be clock-irrelevant.

In particular, $\mathsf{D}^{\forall} A \simeq \mathscr{D}(A + \mathsf{D}^{\forall} A)$, and so carries a convex algebra structure $(\delta^{\forall}, \oplus^{\forall})$ as well as a map $\mathsf{step}^{\forall} : \mathsf{D}^{\forall} A \to \mathsf{D}^{\forall} A$ defined by $\mathsf{step}^{\forall}(x) = \delta(\mathsf{inr}(x))$. Define also $\delta^{\forall} : X \to \mathsf{D}^{\forall} A$ as $\delta^{\forall} x \triangleq \delta(\mathsf{inl}\, x)$.

Example 26. The geometric process can be defined as an element of $\mathscr{N} \to \mathsf{D}^{\forall}(\mathscr{N})$ as $\mathsf{geo}_p\, n \triangleq \Lambda\kappa.\mathsf{geo}_p^{\kappa} n$. This satisfies the equations

$$\begin{aligned}\mathsf{geo}_p &= (\delta^{\forall} 0) \oplus_p^{\forall} \mathsf{step}^{\forall}(\mathsf{geo}_p(1)) \\ &= (\delta^{\forall} 0) \oplus_p \mathsf{step}^{\forall}((\delta^{\forall} 1) \oplus_p \mathsf{step}^{\forall}(\mathsf{geo}_p(2)))\end{aligned}$$

Lemma 3.4.4. *D^{κ} carries a monad structure whose unit is $\delta^{\forall}$ and where the Kleisli extension $\overline{f} : \mathsf{D}^{\forall}(A) \to \mathsf{D}^{\forall}(B)$ of a map $f : A \to \mathsf{D}^{\forall}(B)$ satisfies*

$$\overline{f}(\mathsf{step}^{\forall} x) = \mathsf{step}^{\forall}(\overline{f}(x)) \qquad \overline{f}(\mu \oplus_p^{\forall} \nu) = \overline{f}(\mu) \oplus_p^{\forall} \overline{f}(\nu))$$

Proof. This is a consequence of [85, Lemma 16]. □

Probability of termination

Unfolding the type equivalence $\mathsf{D}^\forall A \simeq \mathscr{D}(A + \mathsf{D}^\forall A)$ we can define maps out of $\mathsf{D}^\forall A$ by cases. For example, we can define the probability of immediate termination $\mathsf{PT}_0 : \mathsf{D}^\forall A \to [0,1]$, as:

$$\begin{aligned}\mathsf{PT}_0(\delta^\forall a) &= 1\\ \mathsf{PT}_0(\mathsf{step}^\forall d) &= 0\\ \mathsf{PT}_0(x \oplus_p^\forall y) &= p \cdot \mathsf{PT}_0(x) + (1-p) \cdot \mathsf{PT}_0(y).\end{aligned}$$

Likewise, we define a function $\mathsf{run} : \mathsf{D}^\forall A \to \mathsf{D}^\forall A$ that runs a computation for one step, eliminating a single level of $\mathsf{step}^\forall$ operations:

$$\begin{aligned}\mathsf{run}(\delta^\forall a) &= \delta^\forall a\\ \mathsf{run}(\mathsf{step}^\forall d) &= d\\ \mathsf{run}(x \oplus_p^\forall y) &= \mathsf{run}\, x \oplus_p^\forall \mathsf{run}\, y.\end{aligned}$$

We can hence compute the probability of termination in n steps, by first running a computation for n steps, and then computing the probability of immediate termination of the result:

$$\mathsf{PT}_n(x) = \mathsf{PT}_0(\mathsf{run}^n x).$$

For example, running the geometric process for one step gives:

$$\mathsf{run}(\mathsf{geo}_p\, 0) = (\delta^\forall 0) \oplus_p ((\delta^\forall 1) \oplus_p \mathsf{step}^\forall(\mathsf{geo}_p\,(2))),$$

so $\mathsf{PT}_1(\mathsf{geo}_p\, 0) = p + (1-p)p = 2p - p^2$.

Rather than eliminating a full level of $\mathsf{step}^\forall$ operations with run, it is sometimes useful to allow different branches to run for a different number of steps. We capture this in the relation $\rightsquigarrow$:

Definition 3.4.5. Define the relation $\rightsquigarrow$: $\mathsf{D}^\forall A \to \mathsf{D}^\forall A \to \mathsf{Prop}$ inductively by the following rules

$$\frac{}{v \rightsquigarrow v} \qquad \frac{}{\mathsf{step}^\forall v \rightsquigarrow v} \qquad \frac{v \rightsquigarrow v' \qquad v' \rightsquigarrow v''}{v \rightsquigarrow v''} \qquad \frac{v_1 \rightsquigarrow v_1' \qquad v_2 \rightsquigarrow v_2'}{v_1 \oplus_p^\forall v_2 \rightsquigarrow v_1' \oplus_p^\forall v_2'}$$

Remark 27. CCTT lacks higher inductive families as primitive, but the special case of the $\rightsquigarrow$ relation can be encoded by defining a fuelled version $\rightsquigarrow^n$ by induction on n, and existentially quantifying n.

The relation $\rightsquigarrow$ is closely related to run.

Lemma 3.4.6. *Let $v, v' : \mathsf{D}^\forall A$. Then:*

1. *For all* $n : \mathscr{N}$, $\nu \rightsquigarrow \mathsf{run}^n \nu$.

2. *If* $\nu \rightsquigarrow \nu'$ *then, for all* $n : \mathscr{N}$, $\mathsf{run}^n \nu \rightsquigarrow \mathsf{run}^n \nu'$.

3. *If* $\nu \rightsquigarrow \nu'$ *then there exists an* $n : \mathscr{N}$ *such that* $\nu' \rightsquigarrow \mathsf{run}^n \nu$.

Proof. The first two facts are proven by induction on n, the last is proven by induction on $\rightsquigarrow$. □

Unlike run, the relation $\rightsquigarrow$ can run different branches of a probabilistic computation for a different number of steps, as illustrated in the following example.

Example 28. Consider: $\nu = (\mathsf{step}^\forall(\delta^\forall a)) \oplus_p^\forall (\mathsf{step}^\forall(\mathsf{step}^\forall(\delta^\forall b)))$. Then run removes $\mathsf{step}^\forall$ operations from both branches.

$$\begin{aligned}
\mathsf{run}(\nu) &= (\delta^\forall a) \oplus_p^\forall (\mathsf{step}^\forall(\delta^\forall b)) \\
\mathsf{run}^2(\nu) &= (\delta^\forall a) \oplus_p^\forall (\delta^\forall b) \\
\mathsf{run}^n(\nu) &= (\delta^\forall a) \oplus_p^\forall (\delta^\forall b) \text{ for } n \geq 2.
\end{aligned}$$

As per Lemma 3.4.6, we have $\nu \rightsquigarrow (\mathsf{run}^n \nu)$, for each $n : \mathscr{N}$, as well as:

$$\begin{aligned}
\nu &\rightsquigarrow (\delta^\forall a) \oplus_p^\forall (\mathsf{step}^\forall(\mathsf{step}^\forall(\delta^\forall b))) \\
\nu &\rightsquigarrow (\mathsf{step}^\forall(\delta^\forall a)) \oplus_p^\forall (\mathsf{step}^\forall(\delta^\forall b)) \\
\nu &\rightsquigarrow (\mathsf{step}^\forall(\delta^\forall a)) \oplus_p^\forall (\delta^\forall b).
\end{aligned}$$

We will use this flexibility in running different branches for a different number of steps in both our proofs and examples, such as in the proof of Lemma 3.7.8 and Theorem 3.9.4.

We show two more useful properties of the relation $\rightsquigarrow$. Firstly, the bind operation preserves the $\rightsquigarrow$ relation, which follows by an easy induction on $\rightsquigarrow$.

Lemma 3.4.7. *If* $f : A \to \mathsf{D}^\forall(B)$, *and* $\nu, \nu' : \mathsf{D}^\forall A$ *such that* $\nu \rightsquigarrow \nu'$. *Then also* $\overline{f}(\nu) \rightsquigarrow \overline{f}(\nu')$.

And secondly, the probability of termination is monotone with respect to n, and along $\rightsquigarrow$:

Lemma 3.4.8. *For* $\nu, \nu' : \mathsf{D}^\forall A$, *we have:*

1. *For all* $n, m : \mathscr{N}$ *such that* $n \leq m$: $\mathsf{PT}_n(\nu) \leq \mathsf{PT}_m(\nu)$.

2. *If* $\nu \rightsquigarrow \nu'$ *then for all* $n : \mathscr{N}$: $\mathsf{PT}_n(\nu) \leq \mathsf{PT}_n(\nu')$.

Proof. The first statement is by induction on n and case analysis for ν, the second statement by induction on n and $\rightsquigarrow$. □

$$\frac{x:\sigma \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash x:\sigma} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \langle\rangle : 1} \qquad \frac{n:\mathscr{N} \qquad \vdash \Gamma}{\Gamma \vdash \underline{n} : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash L : \mathsf{Nat} \qquad \Gamma \vdash M:\sigma \qquad \Gamma \vdash N:\sigma}{\Gamma \vdash \mathsf{ifz}(L,M,N):\sigma} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{suc}\, M : \mathsf{Nat}} \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{pred}\, M : \mathsf{Nat}}$$

$$\frac{\Gamma \vdash M:\sigma}{\Gamma \vdash \mathsf{inl}\, M : \sigma+\tau} \qquad \frac{\Gamma \vdash M:\tau}{\Gamma \vdash \mathsf{inr}\, M : \sigma+\tau}$$

$$\frac{\Gamma \vdash L : \sigma+\sigma' \qquad \Gamma, x:\sigma \vdash M:\tau \qquad \Gamma, y:\sigma' \vdash N:\tau}{\Gamma \vdash \mathsf{case}(L, x.M, y.N):\tau} \qquad \frac{\Gamma \vdash M:\sigma \qquad \Gamma \vdash N:\tau}{\Gamma \vdash \langle M,N\rangle : \sigma\times\tau}$$

$$\frac{\Gamma \vdash M : \sigma\times\tau}{\Gamma \vdash \mathsf{fst}\, M : \sigma} \qquad \frac{\Gamma \vdash M : \sigma\times\tau}{\Gamma \vdash \mathsf{snd}\, M : \tau} \qquad \frac{\Gamma, (x:\sigma) \vdash M:\tau}{\Gamma \vdash \mathsf{lam}\; x.M : \sigma\to\tau}$$

$$\frac{\Gamma \vdash N : \sigma\to\tau \qquad \Gamma \vdash M:\sigma}{\Gamma \vdash NM:\tau} \qquad \frac{\Gamma \vdash M : \tau[\mu X.\tau/X]}{\Gamma \vdash \mathsf{fold}\, M : \mu X.\tau} \qquad \frac{\Gamma \vdash M : \mu X.\tau}{\Gamma \vdash \mathsf{unfold}\, M : \tau[\mu X.\tau/X]}$$

$$\frac{\Gamma \vdash M:\sigma \qquad \Gamma \vdash N:\sigma \qquad p:(0,1)}{\Gamma \vdash \mathsf{choice}^p(M,N):\sigma}$$

Figure 3.2: Typing rules for $\mathsf{FPC}_\oplus$

3.5 Probabilistic FPC

In this section we define the language $\mathsf{FPC}_\oplus$, its typing rules and operational semantics in CCTT. $\mathsf{FPC}_\oplus$ is the extension of simply typed lambda calculus with recursive types and probabilistic choice. The typing rules of $\mathsf{FPC}_\oplus$ are presented in Figure 3.2. In typing judgements, Γ is assumed to be a variable context and all types (which include recursive types of the form $\mu X.\tau$) are closed.

We assume a given representation of terms and types of $\mathsf{FPC}_\oplus$. For example, these can be represented as inductive types with the notion of closedness defined as decidable properties on these. We will also assume that terms are annotated with enough types so that one can deduce the type of subterms from terms. This allows the evaluation function and the denotational semantics to be defined by induction on terms rather than typing judgement derivations. This also means that the typing judgement $\Gamma \vdash M:\sigma$ is decidable. We write Ty for the type of closed types, and $\mathsf{Tm}^\Gamma_\sigma$ for the type of terms M satisfying $\Gamma \vdash M:\sigma$, constructed as a subtype of the type of all terms. When Γ is empty we write simply Tm_σ. We write Val_σ for the set of closed values of type σ, as captured by the grammar

$$V,W ::= \langle\rangle \mid \underline{n} \mid \mathsf{lam}\; x.M \mid \mathsf{fold}\, V \mid \langle V,W\rangle \mid \mathsf{inl}\, V \mid \mathsf{inr}\, V$$

The operational semantics is given by a function eval^κ of type

$$\mathsf{eval}^\kappa : \{\sigma : \mathsf{Ty}\} \to \mathsf{Tm}_\sigma \to \mathsf{D}^\kappa(\mathsf{Val}_\sigma) \tag{3.6}$$

associating to a term M an element of our representation of a distribution over syntactic values, that is, an element of the free guarded convex delay algebra over Val_σ. The corresponding element of the coinductive convex delay algebra can be defined as

$$\mathsf{eval}(M) \triangleq \Lambda\kappa.\mathsf{eval}^\kappa(M) : \mathsf{D}^\forall(\mathsf{Val}_\sigma)$$

The function eval^κ is defined by an outer guarded recursion and an inner induction on terms, and the cases are given in Figure 3.3. The figure overloads names of some term constructors to functions of values, e.g. $\mathbf{inl} : \mathsf{Val}_\sigma \to \mathsf{Val}_{\sigma+\tau}$ mapping V to $\mathsf{inl}\,V$, and $\mathbf{pred}, \mathbf{suc} : \mathsf{Val}_{\mathsf{Nat}} \to \mathsf{Val}_{\mathsf{Nat}}$ defined using $\mathsf{Val}_{\mathsf{Nat}} \simeq \mathscr{N}$. The figure also uses matching by cases, using, for example, that all values of function type are of the form $\mathsf{lam}\,x.M'$ for some M'.

In the cases of $\mathsf{case}(L, x.M, y.N)$ and MN the recursive calls are under a tick and so can be justified by guarded recursion. This is necessary, because these cannot be justified by induction on terms. Also, the case for $\mathsf{unfold}\,M$ introduces a computation step using step^κ. While this is not strictly necessary to define the operational semantics, we introduce it to synchronise with the steps of the denotational semantics defined in Section 3.6.

In $\mathsf{FPC}_\oplus$ we can define a fixpoint combinator as illustrated below:

Example 29. We define $\cdot \vdash \mathsf{Y} : ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau$ by $\mathsf{Y} \triangleq \mathsf{lam}\,f.\mathsf{lam}\,z.e_f(\mathsf{fold}\,e_f)z$, where

$$e_f : (\mu X.(X \to \sigma \to \tau)) \to \sigma \to \tau$$
$$e_f \triangleq \mathsf{lam}\,y.\mathsf{let}\,y' = \mathsf{unfold}\,y\,\mathsf{in}\,f(\mathsf{lam}\,x.y'yx)$$

Here, $y : \mu X.(X \to \sigma \to \tau)$, $y' : (\mu X.(X \to \sigma \to \tau)) \to \sigma \to \tau$ and $\mathsf{lam}\,x.y'yx : \sigma \to \tau$.

Then, for any values $\cdot \vdash f : (\sigma \to \tau) \to \sigma \to \tau$ and $\cdot \vdash V : \sigma$,

$$\mathsf{eval}^\kappa((\mathsf{Y}f)(V)) = (\Delta^\kappa)^4(\mathsf{eval}^\kappa((f(\mathsf{Y}f))(V)))$$

where $\Delta^\kappa \triangleq (\mathsf{step}^\kappa \circ \mathsf{next}^\kappa)$.

Contextual Refinement

Two terms M, N are contextually equivalent if for any closing context $C[-]$ of ground type the terms $C[M]$ and $C[N]$ have the same observational behaviour. Here, the only observable behaviour we consider is the probability of termination for programs of type 1, which, for a closed term M should be the limit of the sequence $\mathsf{PT}_n(\mathsf{eval}\,M)$. Since this limit can not be computed in finite time, we define contextual approximation using finite approximations.

$$
\begin{aligned}
\mathsf{eval}^\kappa(V) &\triangleq \delta^\kappa V \\
\mathsf{eval}^\kappa(\mathsf{pred}\,M) &\triangleq \mathsf{D}^\kappa(\mathbf{pred})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{suc}\,M) &\triangleq \mathsf{D}^\kappa(\mathbf{suc})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{inl}\,M) &\triangleq \mathsf{D}^\kappa(\mathbf{inl})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{inr}\,M) &\triangleq \mathsf{D}^\kappa(\mathbf{inr})(\mathsf{eval}^\kappa M) \\
\mathsf{eval}^\kappa(\mathsf{ifz}(L,M,N)) &\triangleq \mathsf{eval}^\kappa L >\!\!>\!\!=^\kappa \begin{cases} \underline{0} \mapsto \mathsf{eval}^\kappa(M) \\ \underline{n+1} \mapsto \mathsf{eval}^\kappa(M) \end{cases} \\
\mathsf{eval}^\kappa(\mathsf{fst}\,M) &\triangleq (\mathsf{eval}^\kappa M) >\!\!>\!\!=^\kappa \lambda\langle V,W\rangle.\delta^\kappa V \\
\mathsf{eval}^\kappa(\mathsf{snd}\,M) &\triangleq (\mathsf{eval}^\kappa M) >\!\!>\!\!=^\kappa \lambda\langle V,W\rangle.\delta^\kappa W \\
\mathsf{eval}^\kappa\langle M,N\rangle &\triangleq \mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda V. \\
&\qquad \mathsf{eval}^\kappa N >\!\!>\!\!=^\kappa \lambda W.\delta^\kappa\langle V,W\rangle \\
\mathsf{eval}^\kappa(\mathsf{case}(L,x.M,y.N)) &\triangleq \mathsf{eval}^\kappa(L) \\
&\qquad >\!\!>\!\!=^\kappa \begin{cases} \mathsf{inl}\,V \mapsto \mathsf{step}^\kappa(\lambda\alpha.\mathsf{eval}^\kappa(M[V/x])) \\ \mathsf{inr}\,V \mapsto \mathsf{step}^\kappa(\lambda\alpha.\mathsf{eval}^\kappa(N[V/y])) \end{cases} \\
\mathsf{eval}^\kappa(MN) &\triangleq \mathsf{eval}^\kappa(M) >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\ x.M').\ \mathsf{eval}^\kappa(N) \\
&\qquad >\!\!>\!\!=^\kappa \lambda V.\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\mathsf{eval}^\kappa(M'[V/x])) \\
\mathsf{eval}^\kappa(\mathsf{fold}\,M) &\triangleq \mathsf{D}^\kappa(\mathsf{fold})(\mathsf{eval}^\kappa(M)) \\
\mathsf{eval}^\kappa(\mathsf{unfold}\,M) &\triangleq \mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\,V).\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\delta^\kappa V) \\
\mathsf{eval}^\kappa(\mathsf{choice}^p(M,N)) &\triangleq (\mathsf{eval}^\kappa M) \oplus_p^\kappa (\mathsf{eval}^\kappa N)
\end{aligned}
$$

Figure 3.3: The evaluation function eval^κ.

Definition 3.5.1 (Contextual Refinement)**.** Let $\Gamma \vdash M,N : \tau$ be terms. We say that N *contextually refines* M if for any closing context of unit type $C : (\Gamma \vdash \sigma) \Rightarrow (\cdot \vdash 1)$, and for any m there exists an n such that $\mathsf{PT}_m(\mathsf{eval}(C[M])) \leq \mathsf{PT}_n(\mathsf{eval}(C[N]))$. In this case, write $M \preceq_{\mathsf{Ctx}} N$. We say that M and N are contextually equivalent ($M \equiv_{\mathsf{Ctx}} N$) if $M \preceq_{\mathsf{Ctx}} N$ and $N \preceq_{\mathsf{Ctx}} M$.

Definition 3.5.1 should be read as defining a predicate on M and N in CCTT, *i.e.*, a function $\mathsf{Tm}^\Gamma_\sigma \to \mathsf{Tm}^\Gamma_\sigma \to \mathsf{Prop}$ using the universal and existential quantification in Prop. The notion of closing context $C : (\Gamma \vdash \sigma) \Rightarrow (\cdot \vdash 1)$ is a special case of a typing judgement on contexts $C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$ defined in a standard way. Note that context $C[-]$ may capture free variables in M.

Remark 30. Our definition of contextual refinement, being defined using finite approximations, only approximates the classical notion of contextual refinement, in the sense that if, for example, $C[M]$ terminates immediately and $C[N]$ only terminates in

the limit then they would have the same termination probability in the classical sense, in the limit, but they are not related by finite approximations.

3.6 Denotational semantics

We now define a denotational semantics for $\mathsf{FPC}_{\oplus}$. Specifically we define two functions

$$[\![-]\!]^{\kappa} : \mathsf{Ty} \to \mathsf{U}$$
$$[\![-]\!]^{\kappa}_{-} : \{\Gamma : \mathsf{Ctx}\} \to \{\sigma : \mathsf{Ty}\} \to \mathsf{Tm}^{\Gamma}_{\sigma} \to [\![\Gamma]\!]^{\kappa} \to \mathsf{D}^{\kappa}[\![\sigma]\!]^{\kappa}$$

which interpret types and terms, respectively, and which are defined by the clauses in Figure 3.4. The interpretation of types is defined by guarded recursion. Note that by using guarded recursion, one can avoid having to define the denotation of open types. The interpretation of terms is defined by induction on terms and uses the notation

$$d \cdot e \triangleq d >>=^{\kappa} \lambda f.e >>=^{\kappa} \lambda v.\mathsf{step}^{\kappa}(\lambda \alpha.fv)$$

in the case of function application. This introduces a step, as does the elimination for sum types, and unfolding of terms of recursive types. In the former two such cases, the steps are not necessary for defining the semantics, but are introduced to align with the steps used in the definition of eval^{κ}; this allows us to prove the soundness theorem below. Before stating it, note that all syntactic values are interpreted as semantic values in the sense that we can define a map

$$[\![-]\!]^{\mathsf{Val},\kappa} : \{\sigma : \mathsf{Ty}\} \to \mathsf{Val}_{\sigma} \to [\![\sigma]\!]^{\kappa}$$

satisfying $[\![V]\!]^{\kappa} = \delta^{\kappa}([\![V]\!]^{\mathsf{Val},\kappa})$ for all V.

Theorem 3.6.1 (Soundness)**.** *For any well typed closed expression $\cdot \vdash M : \sigma$ we have that*

$$\mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\,(\mathsf{eval}^{\kappa} M) \equiv [\![M]\!]^{\kappa}$$

Example 31. Recall the fixed point operator Y from Example 29. Since $\mathsf{eval}^{\kappa}(\mathsf{Y}fV) = (\mathsf{step}^{\kappa} \circ \mathsf{next}^{\kappa})^4(\mathsf{eval}^{\kappa}(f(\mathsf{Y}f)V))$ for any values f and V, we also get

$$[\![\mathsf{Y}fV]\!]^{\kappa} = (\mathsf{step}^{\kappa} \circ \mathsf{next}^{\kappa})^4([\![f(\mathsf{Y}f)\,V]\!]^{\kappa})$$

by Theorem 3.6.1. By a direct calculation similar to the one in Example 29 one can also prove

$$[\![\mathsf{Y}fx]\!]^{\kappa}_{x \mapsto v} = (\mathsf{step}^{\kappa} \circ \mathsf{next}^{\kappa})^4([\![f(\mathsf{Y}f)\,x]\!]^{\kappa}_{x \mapsto v}) \tag{3.7}$$

for any value f and semantic value v.

Interpretation of types

$$\begin{aligned}
[\![\mathsf{Nat}]\!]^\kappa &\triangleq \mathbb{N} & [\![1]\!]^\kappa &\triangleq 1\\
[\![\sigma\times\tau]\!]^\kappa &\triangleq [\![\sigma]\!]^\kappa \times [\![\tau]\!]^\kappa & [\![\sigma\to\tau]\!]^\kappa &\triangleq [\![\sigma]\!]^\kappa \to \mathsf{D}^\kappa[\![\tau]\!]^\kappa\\
[\![\sigma+\tau]\!]^\kappa &\triangleq [\![\sigma]\!]^\kappa + [\![\tau]\!]^\kappa & [\![\mu X.\tau]\!]^\kappa &\triangleq \triangleright^\kappa [\![\tau[\mu X.\tau/X]]\!]^\kappa
\end{aligned}$$

Interpretation of terms

$$\begin{aligned}
[\![\langle\rangle]\!]^\kappa_\rho &\triangleq \delta^\kappa(\star)\\
[\![\underline{n}]\!]^\kappa_\rho &\triangleq \delta^\kappa(n)\\
[\![\mathsf{suc}\,M]\!]^\kappa_\rho &\triangleq \mathsf{D}^\kappa(\mathbf{suc})\left([\![\Gamma\vdash M:\mathsf{Nat}]\!]^\kappa_\rho\right)\\
[\![\mathsf{pred}\,M]\!]^\kappa_\rho &\triangleq \mathsf{D}^\kappa(\mathbf{pred})\left([\![\Gamma\vdash M:\mathsf{Nat}]\!]^\kappa_\rho\right)\\
[\![\mathsf{ifz}(t,M,N)]\!]^\kappa_\rho &\triangleq [\![L]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \begin{cases} 0 & \mapsto [\![M]\!]^\kappa_\rho\\ n+1 & \mapsto [\![N]\!]^\kappa_\rho \end{cases}\\
[\![\mathsf{lam}\ x.M]\!]^\kappa_\rho &\triangleq \delta^\kappa\left(\lambda(v:[\![\sigma]\!]_\rho^{\mathsf{Val},\kappa}).[\![M]\!]^\kappa_{\rho.x\mapsto v}\right)\\
[\![MN]\!]^\kappa_\rho &\triangleq [\![M]\!]^\kappa_\rho\cdot[\![N]\!]^\kappa_\rho\\
[\![\langle M,N\rangle]\!]^\kappa_\rho &\triangleq [\![M]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \lambda v.\left([\![N]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \lambda w.(v,w)\right)\\
[\![\mathsf{fst}\,M]\!]^\kappa_\rho &\triangleq \mathsf{D}^\kappa(\mathsf{pr}_1)\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{snd}\,M]\!]^\kappa_\rho &\triangleq \mathsf{D}^\kappa(\mathsf{pr}_2)\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{inl}\,M]\!]^\kappa_\rho &\triangleq \mathsf{D}^\kappa(\mathbf{inl})\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{inr}\,M]\!]^\kappa_\rho &\triangleq \mathsf{D}^\kappa(\mathbf{inr})\left([\![M]\!]^\kappa_\rho\right)\\
[\![\mathsf{case}(L,x.M,y.N)]\!]^\kappa_\rho &\triangleq [\![L]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \begin{cases} \mathbf{inl}\,v\mapsto \mathsf{step}^\kappa(\lambda\alpha.[\![M]\!]^\kappa_{\rho.x\mapsto v})\\ \mathbf{inr}\,v\mapsto \mathsf{step}^\kappa(\lambda\alpha.[\![N]\!]^\kappa_{\rho.y\mapsto v}) \end{cases}\\
[\![\mathsf{fold}\,M]\!]^\kappa_\rho &\triangleq \mathsf{D}^\kappa(\mathsf{next}^\kappa)([\![M]\!]^\kappa_\rho)\\
[\![\mathsf{unfold}\,M]\!]^\kappa_\rho &\triangleq [\![M]\!]^\kappa_\rho >\!\!>\!\!=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda\alpha.\delta^\kappa(v[\alpha]))\\
[\![\mathsf{choice}^p(M,N)]\!]^\kappa_\rho &\triangleq [\![M]\!]^\kappa_\rho \oplus^\kappa_p [\![N]\!]^\kappa_\rho
\end{aligned}$$

Figure 3.4: Denotational semantics for $\mathsf{FPC}_\oplus$.

3.7 Couplings and Lifting relations

The definition of the logical relation between syntax and semantics requires lifting a relation $\mathscr{R} : A \to B \to \mathsf{Prop}$ between pairs of values to a relation $\overline{\mathscr{R}}^{\kappa} : \mathsf{D}^{\kappa}A \to \mathsf{D}^{\forall}B \to \mathsf{Prop}$ between pairs of computations. Note the asymmetry in the type of $\overline{\mathscr{R}}^{\kappa}$. The reason is that the logical relation will be defined by guarded recursion in the first argument, while using arbitrarily deep unfoldings of the term on the right-hand side. We dedicate this section to explaining this construction and its basic properties.

First recall the notion of coupling [16, 78], which provides a way to lift a relation over $A \times B$ to a relation over $\mathscr{D}A \times \mathscr{D}B$.

Definition 3.7.1. Let $\mathscr{R} : A \to B \to \mathsf{Prop}$, and let $\mu : \mathscr{D}A, \nu : \mathscr{D}B$ be finite distributions. An $\mathscr{R}$-coupling between μ and ν is a distribution on the total space of $\mathscr{R}$, *i.e.*, $\rho : \mathscr{D}(\Sigma(a{:}A), (b{:}B).a\,\mathscr{R}\,b)$, whose marginals are μ and ν:

$$\mathscr{D}(\mathsf{pr}_1)(\rho) = \mu \qquad\qquad \mathscr{D}(\mathsf{pr}_2)(\rho) = \nu,$$

where pr_1 and pr_2 are the projection maps $A \xleftarrow{\mathsf{pr}_1} A \times B \xrightarrow{\mathsf{pr}_2} B$. We write $\mathsf{Cpl}_{\mathscr{R}}(\mu,\nu)$ for the type of $\mathscr{R}$-couplings between μ and ν.

A coupling is a joint distribution over $A \times B$ for which $\mathscr{R}$ always holds for any pair of values sampled from it. The construction of $\mu\ \overline{\mathscr{R}}^{\kappa}\ \nu$ generalizes this idea for pairs of computations whose final values might become available at different times. Recall that by Theorem 3.3.7 a distribution $\mu : \mathsf{D}^{\kappa}A \simeq \mathscr{D}(A + \triangleright^{\kappa}(\mathsf{D}^{\kappa}A))$ must be either a distribution of values, one of delayed computations or a combination of the two. The relation $\mu\ \overline{\mathscr{R}}^{\kappa}\ \nu$ should then hold if (1) the values available in μ now can be matched by ν after possibly some computation steps, and (2) the delayed computation part of μ can be matched later, in a guarded recursive step. The matching of values is done via a coupling, and the computation steps are interpreted using $\rightsquigarrow$. In the definition, we leave the inclusions $\mathscr{D}(\mathsf{inl})$ and $\mathscr{D}(\mathsf{inr})$ implicit and simply write, e.g., $\mu : \mathscr{D}A$ for the first case mentioned above.

Definition 3.7.2 (Lifting of Relations)**.** Given a relation $\mathscr{R} : A \to B \to \mathsf{Prop}$ we define its lift $\overline{\mathscr{R}}^{\kappa} : \mathsf{D}^{\kappa}A \to \mathsf{D}^{\forall}B \to \mathsf{Prop}$ as: $\mu\ \overline{\mathscr{R}}^{\kappa}\ \nu$ if one of the following three options is true:

1. $\mu : \mathscr{D}A$ and there exists a $\nu' : \mathscr{D}B$ such that $\nu \rightsquigarrow \nu'$, and an $\mathscr{R}$-coupling $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu,\nu')$.

2. $\mu : \mathscr{D}(\triangleright^{\kappa}\mathsf{D}^{\kappa}A)$ and $\triangleright^{\kappa}(\alpha : \kappa)\ (((\zeta^{\kappa}(\mu))[\alpha])\ \overline{\mathscr{R}}^{\kappa}\ \nu)$.

3. There exist $\mu_1 : \mathscr{D}A, \mu_2 : \mathscr{D}(\triangleright^{\kappa}\mathsf{D}^{\kappa}A), \nu_1 : \mathscr{D}B, \nu_2 : \mathsf{D}^{\forall}B$, and $p : (0,1)$, such that $\mu = \mu_1 \oplus_p \mu_2$ and $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu_1,\nu_1)$, and $\triangleright^{\kappa}(\alpha : \kappa)\ (\zeta^{\kappa}(\mu_2)\,[\alpha]\ \overline{\mathscr{R}}^{\kappa}\ \nu_2)$.

where $\zeta^\kappa : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A) \to \triangleright^\kappa \mathsf{D}^\kappa A$ is defined as

$$\zeta^\kappa(\delta(x)) \triangleq x$$
$$\zeta^\kappa(\mu \oplus_p \nu) \triangleq \lambda(\alpha : \kappa).\zeta^\kappa(\mu)\,[\alpha] \oplus_p^\kappa \zeta^\kappa(\nu)\,[\alpha]$$

We then define $\overline{\mathscr{R}} : \mathsf{D}^\forall A \to \mathsf{D}^\forall B \to \mathsf{Prop}$ as:

$$\mu\,\overline{\mathscr{R}}\,\nu = \forall\kappa.(\mu[\kappa]\,\overline{\mathscr{R}}^\kappa\,\nu).$$

The options (1) and (2) correspond to the degenerate cases where none of μ is delayed, and where all of μ is delayed, respectively.

By the encoding of coinductive types using guarded recursion, one can prove the following.

Lemma 3.7.3. *If A is clock irrelevant, then $\overline{\mathscr{R}}$ is the coinductive solution to the equation that $\mu\,\overline{\mathscr{R}}\,\nu$ if and only if one of the following:*

1. *$\mu : \mathscr{D}A$ and there exist $\nu' : \mathscr{D}B$ such that $\nu \rightsquigarrow \nu'$ and there exists an $\mathscr{R}$-coupling $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu, \nu')$.*
2. *$\mu : \mathscr{D}(\mathsf{D}^\forall A)$ and $\mathsf{run}(\mu)\,\overline{\mathscr{R}}\,\nu$.*
3. *There exist $\mu_1 : \mathscr{D}A, \mu_2 : \mathscr{D}(\mathsf{D}^\forall A), \nu_1 : \mathscr{D}B, \nu_2 : \mathsf{D}^\forall B$, and $p : (0,1)$, such that $\mu = \mu_1 \oplus_p \mu_2$ and $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu_1, \nu_1)$, and $\mathsf{run}(\mu_2)\,\overline{\mathscr{R}}\,\nu_2$.*

The definition of lifting is quite abstract, but in some concrete cases, such as when the lifted relation is equality, it translates to a more concrete property of distributions. In this paper, we will use the following result, which uses liftings to compare termination probabilities of distributions of unit type:

Lemma 3.7.4. *Let $\mu, \nu : \mathsf{D}^\forall 1$, and let $\mathsf{eq}_1 : 1 \to 1 \to \mathsf{Prop}$ be the identity relation, relating the unique element to itself. Then,*

$$\mu\,\overline{\mathsf{eq}_1}\,\nu \Rightarrow \forall n : \mathscr{N}\,\exists m : \mathscr{N}.\mathsf{PT}_n(\mu) \leq \mathsf{PT}_m(\nu).$$

Proof (sketch). Note that $1 \simeq \forall\kappa.1$ so that Lemma 3.7.3 applies. The proof is first by induction on n, then by case analysis of $\mu\,\overline{\mathsf{eq}_1}\,\nu$. We show the induction step $n = n' + 1$, for case 3 of $\mu\,\overline{\mathsf{eq}_1}\,\nu$. That is, there exist $\mu_1 : \mathscr{D}1, \mu_2 : \mathscr{D}(\mathsf{D}^\forall 1)$ such that $\mu = \mu_1 \oplus_p \mu_2$, and there exist $\nu_1 : \mathscr{D}1, \nu_2 : \mathsf{D}^\forall 1$ such that $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathsf{eq}_1}(\mu_1, \nu_1)$, and moreover $\mathsf{run}(\mu_2)\,\overline{\mathsf{eq}_1}\,\nu_2$.

Then $\mathsf{PT}_n(\mu) = p + (1-p)\mathsf{PT}_{n'}(\mathsf{run}(\mu_2))$. From the induction hypothesis for n' and the fact that $\mathsf{run}(\mu_2)\,\overline{\mathsf{eq}_1}\,\nu_2$, we may conclude that there is an m' such that $\mathsf{PT}_{n'}(\mathsf{run}(\mu_2)) \leq \mathsf{PT}_{m'}(\nu_2)$. From $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$ and Lemma 3.4.6(3), we know that there is an m such that $\nu_1 \oplus_p \nu_2 \rightsquigarrow \mathsf{run}^m \nu$. Then by Lemma 3.4.8 we have:

$$\mathsf{PT}_{m+m'}(\nu) = \mathsf{PT}_{m'}(\mathsf{run}^m \nu)$$

$$\frac{\nu \rightsquigarrow \nu' \qquad \mu\, \overline{\mathscr{R}}^{\kappa}\, \nu}{\mu\, \overline{\mathscr{R}}^{\kappa}\, \nu'} \qquad\qquad \frac{\nu \rightsquigarrow \nu' \qquad \mu\, \overline{\mathscr{R}}^{\kappa}\, \nu'}{\mu\, \overline{\mathscr{R}}^{\kappa}\, \nu}$$

$$\frac{(\mathsf{step}^{\kappa}\mu_1) \oplus_p^{\kappa} (\mathsf{step}^{\kappa}\mu_2)\, \overline{\mathscr{R}}^{\kappa}\, \nu}{\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).(\mu_1\,[\alpha]) \oplus_p^{\kappa} (\mu_2\,[\alpha]))\, \overline{\mathscr{R}}^{\kappa}\, \nu} \qquad \frac{\mu_1\, \overline{\mathscr{R}}^{\kappa}\, \nu_1 \qquad \mu_2\, \overline{\mathscr{R}}^{\kappa}\, \nu_2}{(\mu_1 \oplus_p^{\kappa} \mu_2)\, \overline{\mathscr{R}}^{\kappa}\, (\nu_1 \oplus_p^{\forall} \nu_2)}$$

$$\frac{a\, \mathscr{R}\, b}{(\delta^{\kappa} a)\, \overline{\mathscr{R}}^{\kappa}\, (\delta^{\forall} b)} \qquad\qquad \frac{\mu\, \overline{\mathscr{R}}^{\kappa}\, \overline{\nu} \qquad \forall a, b.\, a\, \mathscr{R}\, b \to f(a)\, \overline{\mathscr{S}}^{\kappa}\, g(b)}{\overline{f}(\mu)\, \overline{\mathscr{S}}^{\kappa}\, \overline{g}(\nu)}$$

Figure 3.5: Reasoning principles for $\overline{\mathscr{R}}^{\kappa}$

$$\begin{aligned} &\geq \mathsf{PT}_{m'}(\nu_1 \oplus_p \nu_2) \\ &= p + (1-p)\mathsf{PT}_{m'}(\nu_2) \\ &\geq p + (1-p)\mathsf{PT}_{n'}(\mathsf{run}(\mu_2)) \\ &= \mathsf{PT}_n(\mu), \end{aligned}$$

which is what we needed to show. □

In the remainder of this section, we will prove some useful reasoning principles for $\overline{\mathscr{R}}^{\kappa}$ that we need for relating the syntax and semantics. These are summarized in Figure 3.5, where the double bar indicates that the rule is bidirectional. First we show that $\overline{\mathscr{R}}^{\kappa}$ is invariant with respect to $\rightsquigarrow$-reductions on its second argument.

Lemma 3.7.5. *If $\nu \rightsquigarrow \nu'$ then $\mu\, \overline{\mathscr{R}}^{\kappa}\, \nu$ if and only if $\mu\, \overline{\mathscr{R}}^{\kappa}\, \nu'$. In particular $\mu\, \overline{\mathscr{R}}^{\kappa}\, (\mathsf{step}^{\forall}\nu)$ if and only if $\mu\, \overline{\mathscr{R}}^{\kappa}\, \nu$.*

Proof. Right to left is by transitivity of $\rightsquigarrow$. For left to right, suppose $\mu\, \overline{\mathscr{R}}^{\kappa}\, \nu$. By Lemma 3.4.6 there exists an n such that $\nu' \rightsquigarrow \mathsf{run}^n\nu$, so it suffices to show that $\mu\, \overline{\mathscr{R}}^{\kappa}\, \mathsf{run}^n(\nu)$. This can be proven by guarded recursion. For example in the case of item (3) of Definition 3.7.2, if $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$ then also $\mathsf{run}^n(\nu) \rightsquigarrow \mathsf{run}^n(\nu_1 \oplus_p \nu_2)$ by Lemma 3.4.6, and the latter equals $\nu_1 \oplus_p \mathsf{run}^n\nu_2$ since ν_1 is a distribution of values. By guarded recursion we get

$$\triangleright^{\kappa}(\alpha:\kappa)\ (\zeta^{\kappa}(\mu_2)\,[\alpha]\, \overline{\mathscr{R}}^{\kappa}\, \mathsf{run}^n(\nu_2))$$

and the proof follows. □

The monads D^{κ} and $\mathsf{D}^{\forall}$ both distinguish between the order of steps and probabilistic choice. However, these cannot be distinguished by any lifted relation.

Lemma 3.7.6. *Let $\mathscr{R} : A \to B \to \mathsf{Prop}$.*

1. *If* $\mu_1, \mu_2 : \triangleright^\kappa \mathsf{D}^\kappa A$, $\nu : \mathsf{D}^\forall B$ *then* $(\mathsf{step}^\kappa \mu_1) \oplus_p^\kappa (\mathsf{step}^\kappa \mu_2) \overline{\mathscr{R}}^\kappa \nu$ *iff* $\mathsf{step}^\kappa(\lambda(\alpha : \kappa).(\mu_1\,[\alpha]) \oplus_p^\kappa (\mu_2\,[\alpha]))\,\overline{\mathscr{R}}^\kappa\,\nu$

2. *If* $\mu : \mathsf{D}^\kappa A$ *and* $\nu_1, \nu_2 : \mathsf{D}^\forall B$ *then* $\mu\,\overline{\mathscr{R}}^\kappa\,\mathsf{step}^\forall(\nu_1 \oplus_p^\forall \nu_2)$ *iff* $\mu\,\overline{\mathscr{R}}^\kappa\,(\mathsf{step}^\forall(\nu_1)) \oplus_p^\forall (\mathsf{step}^\forall(\nu_2))$

Proof. The first statement follows from applying Definition 3.7.2(2). The second statement follows from Lemma 3.7.5. □

Liftings are a useful technique to reason about computations because of the way they interact with choice, and the monad structures of D^κ and $\mathsf{D}^\forall$. For example, we get the following.

Lemma 3.7.7. *Let* $\mathscr{R} : A \to B \to \mathsf{Prop}$*, then if* $\mu_1\,\overline{\mathscr{R}}^\kappa\,\nu_1$ *and* $\mu_2\,\overline{\mathscr{R}}^\kappa\,\nu_2$*, then also* $(\mu_1 \oplus_p^\kappa \mu_2)\,\overline{\mathscr{R}}^\kappa\,(\nu_1 \oplus_p^\forall \nu_2)$.

Now we state and show the bind lemma, that allows us to sequence computations related by liftings, and that will be crucial e.g. in the proof of the fundamental lemma in the following section.

Lemma 3.7.8 (Bind Lemma)**.** *Let* $\mathscr{R} : A \to B \to \mathsf{Prop}$ *and* $\mathscr{S} : A' \to B' \to \mathsf{Prop}$.

1. *If* $a\,\mathscr{R}\,b$ *then* $(\delta^\kappa a)\,\overline{\mathscr{R}}^\kappa\,(\delta^\forall b)$.

2. *If* $f : A \to \mathsf{D}^\kappa A'$ *and* $g : B \to \mathsf{D}^\forall B'$ *satisfy* $f(a)\,\overline{\mathscr{S}}^\kappa\,g(b)$ *whenever* $a\,\mathscr{R}\,b$*, then for all* $\mu : \mathsf{D}^\kappa A$ *and* $\nu : \mathsf{D}^\forall B$ *satisfying* $\mu\,\overline{\mathscr{R}}^\kappa\,\nu$*, also* $\overline{f}(\mu)\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu)$.

Proof (sketch). The first statement is trivial by taking $\rho = \delta((a,b))$. For the second statement, suppose that $\mu\,\overline{\mathscr{R}}^\kappa\,\nu$. By the definition of $\overline{\mathscr{R}}^\kappa$, there are three cases to consider. We only show the first case here. The second case is by guarded recursion, and the third case combines the proofs of the first and second cases.

Suppose $\mu : \mathscr{D}A$, $\nu \rightsquigarrow \nu'$, and $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu, \nu')$. We proceed by induction on ρ: If $\rho = \delta((a,b))$ for some $a : A$ and $b : B$ such that $a\,\mathscr{R}\,b$, then $\overline{f}(\mu) = \overline{f}(\delta^\kappa a) = f(a)$ and $\overline{g}(\nu') = \overline{g}(\delta^\kappa b) = g(b)$ are related in $\overline{\mathscr{S}}^\kappa$ by assumption.

If $\rho = \rho_1 \oplus_q \rho_2$, let $\mu_i \triangleq \mathscr{D}(\mathsf{pr}_1)(\rho_i)$ and $\nu_i \triangleq \mathscr{D}(\mathsf{pr}_2)(\rho_i)$ for $i = 1, 2$. By induction $\overline{f}(\mu_i)\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu_i)$ and since

$$\overline{f}(\mu) = \overline{f}(\mu_1 \oplus_p^\kappa \mu_2) = \overline{f}(\mu_1) \oplus_p^\kappa \overline{f}(\mu_2)$$
$$\overline{g}(\nu) = \overline{g}(\nu_1 \oplus_p^\forall \nu_2) = \overline{g}(\nu_1) \oplus_p^\forall \overline{g}(\nu_2)$$

the case follows from Lemma 3.7.7. □

Value relation

$$\frac{}{n \preceq^{\kappa,\mathsf{Val}}_{\mathsf{Nat}} \underline{n}} \qquad \frac{}{\star \preceq^{\kappa,\mathsf{Val}}_{1} \langle\rangle} \qquad \frac{v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V \qquad w \preceq^{\kappa,\mathsf{Val}}_{\tau} W}{(v,w) \preceq^{\kappa,\mathsf{Val}}_{\sigma\times\tau} \langle V,W\rangle} \qquad \frac{v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V}{\mathbf{inl}\, v \preceq^{\kappa,\mathsf{Val}}_{\sigma+\tau} \mathsf{inl}\, V}$$

$$\frac{v \preceq^{\kappa,\mathsf{Val}}_{\tau} V}{\mathbf{inr}\, v \preceq^{\kappa,\mathsf{Val}}_{\sigma+\tau} \mathsf{inr}\, V} \qquad \frac{\forall w, V.w \preceq^{\kappa,\mathsf{Val}}_{\sigma} V \to v(w) \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(M[V/x])}{v \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\, x.M}$$

$$\frac{\triangleright(\alpha:\kappa).(v[\alpha] \preceq^{\kappa,\mathsf{Val}}_{\tau[\mu X.\tau/X]} V)}{v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} \mathsf{fold}\, V}$$

Expression relation

$$\mu \preceq^{\kappa,\mathsf{Tm}}_{\sigma} d \triangleq \mu \overline{\preceq^{\kappa,\mathsf{Val}}_{\sigma}}^{\kappa} d$$

Figure 3.6: Logical Relation.

3.8 Relating syntax and semantics

This section defines two relations between syntax and semantics and shows how these can be used for reasoning about contextual refinement for $\mathsf{FPC}_{\oplus}$ terms. The relations relate values and general computations respectively and have type

$$\begin{aligned} \preceq^{\kappa,\mathsf{Val}}_{\sigma} &: [\![\sigma]\!]^{\kappa} \to \mathsf{Val}_{\sigma} \to \mathsf{Prop} \\ \preceq^{\kappa,\mathsf{Tm}}_{\sigma} &: \mathsf{D}^{\kappa}[\![\sigma]\!]^{\kappa} \to \mathsf{D}^{\forall}(\mathsf{Val}_{\sigma}) \to \mathsf{Prop} \end{aligned}$$

These are defined simultaneously by guarded recursion and induction on σ. The cases are given in Figure 3.6. Note that using Lemma 3.2.1 one can easily show that the sets Val_{σ} are all clock-irrelevant, so that $\mathsf{D}^{\forall}(\mathsf{Val}_{\sigma})$ is indeed a coinductive type as in Proposition 3.4.3. As anticipated, the relation for computations is defined as a lifting of the relation for values. In particular, this allows us to use the reasoning principles associated with it (e.g., Lemmas 3.7.5,3.7.7 and 3.7.8) when using it in proofs and examples.

We can define a relation for pairs of open terms as follows:

Definition 3.8.1. For $M,N : \mathsf{Tm}^{\Gamma}_{\sigma}$ we define

$$M \preceq^{\kappa,\Gamma}_{\sigma} N \triangleq \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \mathsf{eval}(N[\delta])\right)$$

Here δ is a closing substitution for Γ and $\rho : [\![\Gamma]\!]^{\kappa}$ an environment. $\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta$ denotes that for every variable x of Γ the corresponding semantic and syntactic value in ρ and δ are related.

We now state the fundamental lemma, that allows us to relate a term to itself.

Lemma 3.8.2 (Fundamental Lemma)**.** *For all $M \in \mathsf{Tm}^{\Gamma}_{\sigma}$ we have that $M \preceq^{\kappa,\Gamma}_{\sigma} M$.*

Corollary 3.8.3. *For terms $M,N : \mathsf{Tm}^{\Gamma}_{\sigma}$ we have that*

$$(\forall \rho : [\![\Gamma]\!]^{\kappa}.([\![M]\!]^{\kappa}_{\rho} = [\![N]\!]^{\kappa}_{\rho})) \to M \preceq^{\kappa,\Gamma}_{\sigma} N$$

This relation is a congruence, i.e., it is closed under the typing rules. In particular:

Lemma 3.8.4 (Congruence Lemma)**.** *For any terms $M,N \in \mathsf{Tm}^{\Gamma}_{\sigma}$ and every context $C : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$ we get that*

$$M \preceq^{\kappa,\Gamma}_{\sigma} N \to C[M] \preceq^{\kappa,\Delta}_{\tau} C[N]$$

For related terms of unit type, we can obtain an inequality between their probabilities of termination (c.f. Lemma 3.8.5). First, we need the following preliminary lemma:

Lemma 3.8.5. *For any $M,N \in \mathsf{Tm}_1$ we have that*

$$(\forall \kappa.[\![M]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{1} \mathsf{eval}(N)) \to (\mathsf{eval}(M)\,\overline{\mathsf{eq}_1}\,\mathsf{eval}(N))$$

Proof. Assume that $\forall \kappa.[\![M]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{1} \mathsf{eval}(N)$ and let $\kappa : \mathsf{clock}$ be arbitrary. We need to show that $(\mathsf{eval}^{\kappa}(M)\,\overline{\mathsf{eq}_1}^{\kappa}\,\mathsf{eval}(N))$. Since Theorem 3.6.1 implies $[\![M]\!]^{\kappa} = \mathsf{D}^{\kappa}[\![-]\!]^{\mathsf{Val},\kappa}(\mathsf{eval}^{\kappa}(M))$ we directly get that $\mathsf{D}^{\kappa}([\![-]\!]^{\mathsf{Val},\kappa})(\mathsf{eval}^{\kappa}(M)) \preceq^{\kappa,\mathsf{Tm}}_{1} \mathsf{eval}(N)$. On the type 1, the function $[\![-]\!]^{\mathsf{Val},\kappa}$ is an isomorphism, and, up to this, $\preceq^{\kappa,\mathsf{Val}}_{1}$ is simply eq_1, so we conclude $(\mathsf{eval}(M)\,\overline{\mathsf{eq}_1}\,\mathsf{eval}(N))$ as desired. □

From the results above, we derive our main theorem, stating that the logical relation is sound with respect to contextual refinement:

Theorem 3.8.6. *For any terms $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$ we have*

$$(\forall \kappa.M \preceq^{\kappa,\Gamma}_{\sigma} N) \to M \preceq_{\mathsf{Ctx}} N$$

Proof. Let M,N be arbitrary and $\forall \kappa.M \preceq^{\kappa,\Gamma}_{\sigma} N$. It remains to show that $M \preceq_{\mathsf{Ctx}} N$. To do so, let $C : (\Gamma \vdash \sigma) \Rightarrow (\cdot \vdash 1)$, then Lemma 3.8.4 implies that $\forall \kappa.C[M] \preceq^{\kappa,\cdot}_{1} C[N]$ which by definition is equivalent to $\forall \kappa.[\![C[M]]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{1} \mathsf{eval}(C[N])$. By Lemma 3.8.5 it follows that $(\mathsf{eval}(C[M]))\ \overline{\mathsf{eq}_1}\ (\mathsf{eval}(C[N]))$ and Lemma 3.7.4 completes the proof. □

3.9 Examples

The following is an immediate consequence of Theorem 3.8.6.

Theorem 3.9.1. *Let M and N be closed terms of the same type σ. If $\mathsf{eval}\,M \rightsquigarrow \mathsf{eval}\,N$, then $M \equiv_{\mathsf{Ctx}} N$.*

Proof. By Lemma 3.7.5, $\mu \preceq_\sigma^{\kappa,\mathsf{Val}} \mathsf{eval}\,M$ if and only if $\mu \preceq_\sigma^{\kappa,\mathsf{Val}} \mathsf{eval}\,N$. By Lemma 3.8.2 $[\![M]\!]^\kappa \preceq_\sigma^{\kappa,\mathsf{Val}} \mathsf{eval}\,M$, so $[\![M]\!]^\kappa \preceq_\sigma^{\kappa,\mathsf{Val}} \mathsf{eval}\,N$, and so by Theorem 3.8.6 also $M \preceq_{\mathsf{Ctx}} N$. The other way is similar. □

For example, since $\mathsf{eval}(\mathsf{Y}fV) \rightsquigarrow \mathsf{eval}(f(\mathsf{Y}f)V)$, also $\mathsf{Y}fV \equiv_{\mathsf{Ctx}} f(\mathsf{Y}f)V$. Similarly, since

$$\mathsf{eval}((\mathsf{lam}\ x.M)V) = \mathsf{step}^\forall(\mathsf{eval}(M[V/x])) \rightsquigarrow \mathsf{eval}(M[V/x])$$
$$\mathsf{eval}(\mathsf{unfold}\,(\mathsf{fold}\,V)) = \mathsf{step}^\forall(V) \rightsquigarrow V$$

for closed values V and $\mathsf{lam}\ x.M$, we get the usual call-by-value β rules up to contextual equivalence.

A hesitant identity function

For any type σ and rational number $0 < p < 1$ we define the hesitant identity function $\mathsf{hid}_\mathsf{p} : \sigma \to \sigma$ as

$$\mathsf{hid}_\mathsf{p} \triangleq \mathsf{lam}\ z.\big(\big(\mathsf{Y}(\mathsf{hid}'_\mathsf{p})\big)\,z\big) \qquad \text{where}$$
$$\mathsf{hid}'_\mathsf{p} \triangleq \mathsf{lam}\ f.\mathsf{lam}\ x.\mathsf{choice}^p(x, fx)$$

We will show that $\mathsf{hid}_\mathsf{p} \preceq_{\mathsf{Ctx}} \mathsf{id}$. Note that the $\mathsf{id} \preceq_{\mathsf{Ctx}} \mathsf{hid}_\mathsf{p}$ is not true for the reason mentioned in Remark 30. Since hid_p is a value, it suffices to show that

$$[\![\mathsf{hid}_\mathsf{p}]\!]^{\mathsf{Val},\kappa}(v) \preceq_\sigma^{\kappa,\mathsf{Val}} V \tag{3.8}$$

for all v, V satisfying $v \preceq_\sigma^{\kappa,\mathsf{Val}} V$. We note the following lemma which can be proved by a small calculation using Lemma 3.7.6. Recall the notation $\Delta^\kappa \triangleq \mathsf{step}^\kappa \circ \mathsf{next}^\kappa$.

Lemma 3.9.2. *For any value v and any distribution ν, the statement $[\![\mathsf{hid}_\mathsf{p}]\!]^{\mathsf{Val},\kappa}(v) \preceq_\sigma^{\kappa,\mathsf{Val}} \mu$ is equivalent to*

$$(\triangleright^\kappa)^5\left(((\Delta^\kappa)^2(v)) \oplus_p^\kappa ([\![\mathsf{hid}_\mathsf{p}]\!]^{\mathsf{Val},\kappa}(v)) \preceq_\sigma^{\kappa,\mathsf{Val}} \mu\right)$$

Using this, (3.8) can be proved using guarded recursion and Lemma 3.7.7.

A fair coin from an unfair coin

Define $\mathsf{bool} \triangleq 1+1$ referring to the elements as tt and ff. A *coin* is a program of the form $\mathsf{choice}^p(\mathsf{tt},\mathsf{ff}) : \mathsf{bool}$ for $p : (0,1)$. A coin is called *fair* if $p = \frac{1}{2}$.

Using a recursive procedure, we can encode a fair coin $\mathsf{efair}_\mathsf{p}\langle\rangle$ from an unfair coin as follows.

$$\mathsf{efair}_\mathsf{p} \triangleq \mathsf{Y}(\mathsf{efair}'_\mathsf{p}) : 1 \to \mathsf{bool}$$

$$\begin{aligned}\mathsf{efair}'_{\mathsf{p}} \triangleq \mathsf{lam}\ g.\mathsf{lam}\ z.&\mathsf{let}\ x = \mathsf{choice}^{p}(\mathsf{tt},\mathsf{ff})\\ &\mathsf{let}\ y = \mathsf{choice}^{p}(\mathsf{tt},\mathsf{ff})\\ &\mathsf{if}\ \mathsf{eqbool}(x,y)\ \mathsf{then}\ g(z)\ \mathsf{else}\ x\end{aligned}$$

Because this program only terminates in the limit, we cannot prove it equivalent to a fair coin. Instead, we will prove it equivalent to a hesitant fair coin defined as follows.

$$\mathsf{hfair}_{\mathsf{p}} \triangleq (\mathsf{hid}_{\mathsf{p}}(\mathsf{choice}^{\frac{1}{2}}(\mathsf{tt},\mathsf{ff})))$$

Theorem 3.9.3. $\mathsf{efair}_{\mathsf{p}}\langle\rangle$ *is contextually equivalent to* $\mathsf{hfair}_{2\cdot\mathsf{p}(1-\mathsf{p})}$.

Proof. Unfolding definitions shows that

$$\begin{aligned}&\mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle)\\ &\rightsquigarrow (\mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle) \oplus^{\forall}_{p} \mathsf{tt}) \oplus^{\forall}_{p} (\mathsf{ff} \oplus^{\forall}_{p} \mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle))\\ &= (\mathsf{tt} \oplus^{\forall}_{2p(1-p)} (\mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle))) \oplus^{\forall}_{\frac{1}{2}} (\mathsf{ff} \oplus^{\forall}_{2p(1-p)} (\mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle)))\end{aligned}$$

On the other hand, $[\![\mathsf{hfair}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^{\kappa}$ equals

$$\Delta^{\kappa}([\![\mathsf{hid}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^{\mathsf{Val},\kappa}(\mathsf{tt}) \oplus^{\kappa}_{\frac{1}{2}} [\![\mathsf{hid}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^{\mathsf{Val},\kappa}(\mathsf{ff}))$$

We show that $[\![\mathsf{hfair}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle)$ by guarded recursion. So assume $\triangleright^{\kappa}\left([\![\mathsf{hfair}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle)\right)$. By the above and Lemmas 3.7.5 and 3.7.7 it suffices to show

$$\triangleright^{\kappa}\left([\![\mathsf{hid}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^{\kappa}(\mathsf{tt}) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} (\mathsf{tt} \oplus^{\forall}_{2p(1-p)} (\mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle)))\right)$$

and similarly for ff. By Lemmas 3.9.2 and 3.7.7, this reduces to showing $\triangleright^{\kappa}(\mathsf{tt} \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{tt})$ and $\triangleright^{\kappa}([\![\mathsf{hid}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^{\kappa}(\mathsf{tt}) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{efair}_{\mathsf{p}}\langle\rangle))$. The former of this follows from Lemma 3.8.2 and the latter is the guarded recursion hypothesis. The other direction is similar. □

Relating hesitant identity functions

As an example of reasoning about programs with different convergence speeds, we show the following.

Theorem 3.9.4. *For any* $p,q \in (0,1)$ *the programs* $\mathsf{hid}_{\mathsf{p}}$ *and* $\mathsf{hid}_{\mathsf{q}}$ *are contextually equivalent, as are* $\mathsf{hfair}_{\mathsf{p}}$ *and* $\mathsf{hfair}_{\mathsf{q}}$.

Proof (sketch). We show that $[\![\mathsf{hid}_{\mathsf{p}}]\!]^{\kappa} \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\sigma} \mathsf{eval}(\mathsf{hid}_{\mathsf{q}})$. Since $\mathsf{hid}_{\mathsf{p}}$ is a value, this means showing $[\![\mathsf{hid}_{\mathsf{p}}]\!]^{\mathsf{Val},\kappa}(v) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \mathsf{eval}(\mathsf{hid}_{\mathsf{q}}\,V)$ for all v,V such that $v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V$. We prove this by guarded recursion. First, suppose $p<q$. Since $\mathsf{eval}(\mathsf{hid}_{\mathsf{q}}\,V)$ is related in the symmetric transitive closure of $\rightsquigarrow$ to

$$V \oplus^{\forall}_{q} \mathsf{eval}(\mathsf{hid}_{\mathsf{q}}\,V) = V \oplus^{\forall}_{p} (V \oplus^{\forall}_{\frac{q-p}{1-p}} \mathsf{eval}(\mathsf{hid}_{\mathsf{q}}\,V))$$

by Lemma 3.9.2 it suffices to show $(\triangleright^\kappa)^7(v \preceq^{\kappa,\mathsf{Val}}_\sigma V)$ and

$$(\triangleright^\kappa)^5\left([\![\mathsf{hid}_\mathsf{p}]\!]^{\mathsf{Val},\kappa}(v) \preceq^{\kappa,\mathsf{Val}}_\sigma (V \oplus^\forall_{\frac{q-p}{1-p}} \mathsf{eval}(\mathsf{hid}_\mathsf{q} V))\right)$$

The latter follows from the guarded recursion hypothesis as well as $[\![\mathsf{hid}_\mathsf{p}]\!]^\kappa(v) \preceq^{\kappa,\mathsf{Val}}_\sigma V$.

In the case of $p > q$, an easy induction on n shows that $\mathsf{eval}(\mathsf{hid}_\mathsf{q} V)$ is related in the transitive symmetric closure of $\rightsquigarrow$ to

$$V \oplus^\forall_{\sum_{k=0}^{n-1} q(1-q)^k} (\mathsf{eval}(\mathsf{hid}_\mathsf{q} V))$$

Since there exists an n such that $p < \sum_{k=0}^{n-1} q(1-q)^k$, the claim now follows as in the previous case.

Since $[\![\mathsf{hfair}_\mathsf{p}]\!]^\kappa = \Delta^\kappa([\![\mathsf{hid}_\mathsf{p}]\!]^\kappa(\mathsf{tt})) \oplus^\kappa_{\frac{1}{2}} \Delta^\kappa([\![\mathsf{hid}_\mathsf{p}]\!]^\kappa(\mathsf{ff}))$ and

$$\mathsf{eval}(\mathsf{hfair}_\mathsf{q}) = \mathsf{eval}(\mathsf{hid}_\mathsf{q}\,\mathsf{tt}) \oplus^\forall_{\frac{1}{2}} \mathsf{eval}(\mathsf{hid}_\mathsf{q}\,\mathsf{ff})$$

the second statement follows from $[\![\mathsf{hid}_\mathsf{p}]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\sigma} \mathsf{eval}(\mathsf{hid}_\mathsf{q})$. □

3.10 Related Work

We have discussed some related work in the Introduction and throughout the paper; here we discuss additional related work.

In this paper, we have shown how to use synthetic guarded domain theory (SGDT) to model $\mathsf{FPC}_\oplus$. SGDT has been used in earlier work to model PCF [91], a call-by-name variant of FPC [82], FPC with general references [102, 103], untyped lambda calculus with nondeterminism [84], and guarded interaction trees [52]. Thus the key new challenge addressed in this paper is the modelling of probabilistic choice, in combination with recursion, which led us to introduce (guarded) convex delay algebras.

Our logical relation between denotational semantics and operational semantics is inspired by the earlier work on nondeterminism in [84]. Here we use the relation to reason about contextual refinement whereas in *op. cit.* it is used for proving congruence of an applicative simulation relation defined on operational semantics.

Applicative bisimulation for a probabilistic, call-by-value version of PCF is studied in [41]. This work defines an approximation-based operational semantics, where a term evaluates to a family of finite distributions over values, each element of the family corresponding to the values observed after a given finite number of steps. This is reminiscent of our semantics, although in *op. cit.* the approximation semantics is then used to define a limit semantics using suprema.

Probabilistic couplings and liftings have been a popular technique in recent years to reason about probabilistic programs [16, 17], since they provide a compositional way to lift relations from base types to distributions over those types. To the best of

our knowledge, our presentation is the first that uses constructive mathematics in the definition of couplings and the proofs of their properties, e.g. the bind lemma. Our definition of couplings is also asymmetric to account for the fact that the distribution on the left uses guarded recursion. This is similar in spirit to the left-partial coupling definition of [59], where asymmetry is used to account for step indexing.

In this paper, we model the probabilistic effects of $\mathsf{FPC}_{\oplus}$ using guarded / coinductive types. In spirit, this is similar to how effects are being modeled in both coinductive and guarded interaction trees [52, 112]. Coinductive interaction trees are useful for giving denotational semantics for first-order programs and are defined using ordinary type theory, whereas guarded interaction trees can also be applied to give denotational semantics for higher-order programs, but are defined using a fragment of guarded type theory. Interaction trees have recently been extended to account for nondeterminism [35] but, to the best of our knowledge, interaction trees have so far not been extended to account for probabilistic effects.

3.11 Conclusion and Future Work

We have developed a notion of (guarded) convex delay algebras and shown how to use it to define and relate operational and denotational semantics for $\mathsf{FPC}_{\oplus}$ in guarded type theory. To the best of our knowledge, this is the first constructive type theoretic account of the semantics of $\mathsf{FPC}_{\oplus}$.

Future work includes (1) combining and extending the present work with the account of nondeterminism in [84] and to compare the resulting model with the recent classically defined operationally-based logical relation in [7]; and (2) to extend the logical relation to account for approximate relational reasoning (up to a small ε), which would allow us, *e.g.*, to show that constant functions are refinements of their approximations.

Chapter 4

`mitten` : a flexible multimodal proof assistant

Abstract

Recently, there has been a growing interest in type theories which include *modalities*, unary type constructors which need not commute with substitution. Here we focus on MTT [58], a general modal type theory which can internalize arbitrary collections of (dependent) right adjoints [25]. These modalities are specified by mode theories [75], 2-categories whose objects corresponds to modes, morphisms to modalities, and 2-cells to natural transformations between modalities. We contribute a defunctionalized NbE algorithm which reduces the type-checking problem for MTT to deciding the word problem for the mode theory. The algorithm is restricted to the class of *preordered* mode theories—mode theories with at most one 2-cell between any pair of modalities. Crucially, the normalization algorithm does not depend on the particulars of the mode theory and can be applied without change to any preordered collection of modalities. Furthermore, we specify a bidirectional syntax for MTT together with a type-checking algorithm. We further contribute `mitten`, a flexible experimental proof assistant implementing these algorithms which supports all decidable preordered mode theories without alteration.

4.1 Introduction

A fundamental benefit of using type theory is the possibility of working within a proof assistant, which can check and even aid in the construction of complex theorems. Implementing a proof assistant, however, is a highly nontrivial task. In addition to a solid theoretical foundation for the particular type theory, numerous practical implementation issues must be addressed.

Recently, interest has gathered around type theories with *modalities*, unary type constructors which need not commute with substitution. Unfortunately, the situation for modal type theories is even more fraught; the theory for modalities is poorly

understood in general, and it is unknown whether standard implementation techniques extend to support them.

Despite these challenges, mainstream proof assistants have begun to experiment with modalities [109], but these implementations are costly and only apply to a particular modal type theory. In practice, a type theorist may use a particular collection of modalities for only one proof or construction and it is impractical to invest in a specialized modal proof assistant each time. This churn has pushed type theorists to define *general* modal type theories which can be instantiated to a variety of modal situations [58, 76].

We choose to focus on MTT [58], a general modal type theory which can internalize an arbitrary collection of modalities so long as they behave like *right adjoints* [25]. Despite limiting consideration to right adjoints, MTT can be used to model a variety of existing modal type theories including calculi for guarded recursion, internalized parametricity, and axiomatic cohesion. Better still, MTT has a robustly developed metatheory [54, 58] which applies *irrespective* of the chosen modalities. An implementation of MTT could therefore conceivably be designed to allow the user to freely change the collection of modalities without re-implementing the entire proof assistant each time. This, in turn, enables the kind of specialized modal proof assistants previously impractical for one-off modal type theories.

MTT: a general modal type theory

As mentioned, MTT can be instantiated with a collection of modalities. More precisely, MTT is parameterized by a mode theory, a strict 2-category which describes a modal situation. Intuitively, objects (m, n, o) of this mode theory represent distinct type theories which are then connected by 1-cells (μ, ν, ξ) which describe the modalities. The categorical structure ensures that modalities compose and that there is an identity modality. In order to describe more intricate connections and structure, the mode theory also contains 2-cells (α, β). A 2-cell induces a 'natural transformation' between modalities. By carefully choosing 2-cells we can force a modality to e.g. become a comonad, a monad, or an adjoint.

To give a paradigmatic example, consider the mode theory $\mathscr{M}$ with a single object m, a single non-identity morphism $\mu : m \longrightarrow m$ and a 2-cell $\varepsilon : \mu \longrightarrow \mathsf{id}_m$ subject to the equations $\mu \circ \mu = \mu$ and $\varepsilon \star \mu = \mu \star \varepsilon$. This description defines $\mathscr{M}$ as a 2-category with a strictly idempotent comonad μ. Instantiating MTT with this mode theory yields a modality $\langle \mu \mid - \rangle$ together with definable operations shaping $\langle \mu \mid - \rangle$ into an idempotent comonad:

$$\mathsf{extract}_A : \langle \mu \mid A \rangle \to A \qquad \mathsf{dup}_A : \langle \mu \mid A \rangle \simeq \langle \mu \mid \langle \mu \mid A \rangle \rangle$$

Even this simple modal type theory is quite useful; it can serve as a replacement for the experimental version of Agda [109] used to formalize a construction of univalent universes [77].

Given the generality, it is natural to wonder whether instantiating MTT yields a calculus which is feasible to work with in practice. Fortunately, prior Fitch-style type

theories have been highly workable [14, 15, 108] and this trend has continued with MTT [54, 56, 58].

From theory to practice

Unfortunately, converting the theoretical guarantee of normalization into an executable program is not a small step. A first obstacle is the syntax of MTT itself: prior work has exclusively considered an algebraic presentation of the syntax as a generalized algebraic theory. While mathematically elegant, a proof assistant requires a more streamlined and ergonomic syntax. Once a more convenient syntax has been designed, one must adapt the normalization proof to a normalization algorithm. Normalization is proven by a sophisticated *gluing* argument, and while the proof is reminiscent of normalization-by-evaluation [3] it remains to extract such an algorithm. Finally, the normalization algorithm does not give any insight into representing common mode theories or solving their word problems.

Restriction to preordered mode theories Many difficulties flow not from the modalities per se, but from the 2-cells of our mode theory, which induce a new primitive type of substitutions. During normalization these *key substitutions* accumulate at variables. Unfortunately, they disrupt a crucial property of modern NbE algorithms: variables can no longer be presented in a way that is invariant under weakening. Therefore, we restrict our attention to mode theories that are *preordered*, with at most one 2-cell between any pair of modalities.

This allows us to present a syntax that never talks about 2-cells and relies entirely on the elaboration procedure to insert and check 2-cells. In addition to avoiding annotations, this simplifies the normalization algorithm since the troublesome key substitutions trivialize.

Although such a restriction does preclude some examples, preordered mode theories are still expressive enough to model guarded recursion together with an *everything now* modality similar to the one introduced by Clousten et al. [36].

A surface syntax for MTT As a generalized algebraic theory, MTT is presented with explicit substitutions and fully annotated connectives [58]. In order to avoid this bureaucracy, we introduce a bidirectional version of MTT which allows a user of `mitten` to omit almost all type annotations [40].

Normalization-by-evaluation The normalization proof for MTT follows the structure of a normalization-by-evaluation proof. Rather than fixing a rewriting system, a term is *evaluated* into a carefully chosen semantics equipped with a quotation function reifying an element of the semantic domain to a normal form. The entire normalization function is then a round-trip from syntax to semantics and then back to normal forms. While the proof of normalization uses a traditional denotational model for a semantic domain, this approach is unsuitable for implementation.

Instead `mitten` follows the literature on normalization-by-evaluation and uses a *defunctionalized* and *syntactic* semantic domain [3]. This approach has previously been adapted to work with particular a modal type theories [57, 63].

Mode theories As mentioned previously, normalization for MTT does not immediately imply the decidability of type equality. Terms (and therefore types) mention both 1- and 2-cells from the mode theory, and deciding the mode theory is a necessary precondition for deciding type equality. Moreover, deciding the equality of 1- and 2-cells, even in a finitely presented 2-category, is well-known to be undecidable.[1] For us, this situation is slightly improved since for preordered mode theories at least 2-cell equality is trivial. Unfortunately, the undecidability of 1-cell equality remains. Special attention is therefore necessary for each mode theory to ensure that the normalization algorithm for MTT is sufficient to yield a type-checker.

While this rules out a truly generic proof assistant for MTT which works regardless of the choice of mode theory, `mitten` shows that the best theoretically possible result is obtainable. We implement `mitten` to be parameterized by a module describing the mode theory so that the type-checker relies only on the existence of such a decision procedure. In particular, there is no need to alter the entire proof assistant when changing the mode theory; only a new mode theory module is necessary. Crucially, while the user must write a small amount of code, no specialized knowledge of proof assistants is required.

We have implemented several mode theories commonly used with MTT in this way, showing that in practice decidability is no real obstacle. For instance, we have configured `mitten` to support guarded recursion with a combination of two modalities $\Box$ and $\blacktriangleright$. This is the first proof assistant to support this combination of modalities.

Contributions

We contribute a bidirectional syntax for MTT (restricted to preordered mode theories) together with a defunctionalized normalization-by-evaluation algorithm which reduces the type-checking problem to deciding the word problem for the mode theory. We have put these results into practice with `mitten`, a prototype implementation of MTT based on this algorithm. `mitten` also supports the replacement of the underlying mode theory with minimal alterations, allowing a user to construct specialized proof assistants for modal type theories by merely supplying a single module specifying the mode theory together with equality functions for 0-, 1-, and 2-cells.

In Section 4.2 we provide a guided tour of MTT. This section also introduces the bidirectional syntax for MTT and shows how even in this general setting the modalities introduce minimal overhead. Section 4.3 introduces the defunctionalized normalization algorithm for non-specialists and Sections 4.4 and 4.5 completes the description of the core components of `mitten` by describing the type-checking al-

[1] The word problem is well-known to be undecidable for finitely presented groups which can be realized as finitely-presented categories and therefore locally discrete finitely-presentable 2-categories.

gorithm. In so doing, we also describe the novel interface `mitten` uses to represent modalities and show how this interface is implemented.

In Section 4.6 we discuss the realization of mode theories with a representative example: guarded recursion. As previously mentioned, this is the first proof assistant able to support this pair of modalities simultaneously.

4.2 A surface syntax for MTT

Prior to specifying a type-checking algorithm for MTT, we must specify the surface syntax for the language. This question is not satisfactorily addressed in the prior work on MTT; the *generalized algebraic* version of syntax is too verbose to be workable, but the informal pen-and-paper syntax which omits all type annotations cannot be type-checked. Our surface syntax is formulated with an eye towards the type-checking algorithm we will eventually use: a version of Coquand's semantic type-checker [40]. In particular, we will employ a bidirectional surface syntax which minimizes the number of mandatory annotations while still ensuring the decidability of type-checking.

To a first approximation, the surface syntax is divided into two components: checkable and synthesizable terms. Checkable terms include introduction forms while synthesizable terms include elimination forms and variables. By carefully controlling where checkable and synthesizable terms are used, we thereby avoid unnecessary type annotations.

We present the grammar for surface syntax in Section 4.2. While we will defer presenting the actual type-checking algorithm until Section 4.5, in order to make this account as self-contained as possible we provide an example-based introduction to MTT in Section 4.2.

Bidirectional Syntax

As previously mentioned, MTT is parameterized by a mode theory [75] which specifies the modes and modalities of the type theory. We begin by more precisely defining a mode theory in our situation.

Definition 4.2.1. A mode theory is a category whose objects m, n, o we refer to as *modes* and whose morphisms μ, ν we refer to as *modalities*. We further require that each hom-set be equipped with a pre-order $\leq$ compatible with composition. Explicitly, given $\mu, \nu \in \mathsf{Hom}(m, n)$ and $\rho, \sigma \in \mathsf{Hom}(n, o)$ with $\mu \leq \nu$ and $\rho \leq \sigma$ we require $\rho \circ \mu \leq \sigma \circ \nu$.

Equivalently, a mode theory is a preorder-enriched category.

For the remainder of this subsection, we fix a mode theory $\mathscr{M}$. The grammar of the surface syntax is presented below:

$$
\begin{array}{llll}
\textit{(Checkable)} & A,M,N,C & ::= & R \mid (\mu \mid A) \to B \mid A \times B \mid \texttt{Nat} \mid \texttt{U} \mid \lambda(M) \mid (M,N) \\
& & & \mid \texttt{zero} \mid \texttt{succ}(M) \mid \langle \mu \mid A \rangle \mid \texttt{mod}_\mu(M) \\
& & & \mid \texttt{Id}_A(M,N) \mid \texttt{refl}_M \\
\textit{(Synthesizable)} & R,S & ::= & M : A \mid \mathbf{q}_k \mid R(M)_\mu \mid \texttt{pr}_1(R) \mid \texttt{pr}_2(R) \\
& & & \mid \texttt{let}_\mu\ \texttt{mod}_\nu(\_) \leftarrow R\ \texttt{in}\ M\ \texttt{over}\ C \\
& & & \mid \texttt{rec}(C, M_{\mathsf{zero}}, M_{\mathsf{suc}}, N) \mid \texttt{J}(C, c_{\mathsf{refl}}, M)
\end{array}
$$

As mentioned previously, checkable terms consist essentially of introduction forms while synthesizable terms are elimination principles. For instance, the presentation of dependent sums above includes $A \times B$ and (M,N) as checkable terms while $\texttt{pr}_i(R)$ is synthesizable.

By stratifying terms in this way we ensure that annotations are required exactly where ambiguity would arise during type-checking. For instance, this stratification prevents unannotated β-redexes from occurring. Consider again the case of dependent sums. In order to apply a projection to an element (M,N) of dependent sum type, the element must be synthesizable. However, since (M,N) is checkable, the only way to represent $\texttt{pr}_1((M,N))$ in this discipline is to promote (M,N) to a synthesizable term by annotating it: $(M,N) : A \times B$.

Remark 32. In particular, terms in β-normal and η-long form fit into this surface syntax with no additional annotations. Consequently, the normalization theorem for MTT [54] ensures that any term is convertible to one expressible in the surface syntax.

Remark 33. We have made a concession to simplicity and used de Bruijn indices for variables rather than names. This makes the normalization and type-checking algorithms far easier to specify and it is well-known how to pass between syntax with named variables and de Bruijn indices. We will use named variables in examples e.g., $\texttt{let}_\mu\ \texttt{mod}_\nu(y) \leftarrow R\ \texttt{in}\ M\ \texttt{over}\ x.C$ or $(\mu \mid x : A) \to B$ for modal elimination and dependent products respectively.

The surface syntax by example

We will crystallize when a term in the surface syntax is well-formed in Section 4.5 when presenting the type-checking algorithm. In order to cultivate intuition for the theory before this, we will now work through several examples in the language.

Remark 34. We refer the reader to Gratzer et al. [58] for a long form explanation of MTT.

MTT with one mode and one generating modality

Consider MTT instantiated with the mode theory with one mode m and one modality ϕ with no non-trivial equations or inequalities. Then each modality μ is uniquely expressible as ϕ^n, the composition of n copies of ϕ. Just as in ordinary type theory, MTT then has dependent sums, natural numbers, identity types, and their behavior is unchanged.

Unlike in ordinary type theory, each variable is annotated with a modality $x : (\mu \mid A)$ (pronounced $x : A$ annotated by μ). Variables annotated with the identity modality behave like 'ordinary' variables; they can be used freely when working with e.g. natural numbers. Conversely, variables annotated with ϕ^{n+1} cannot be used except in the construction of an element the modal type $\langle \phi \mid A \rangle$.

An element of $\langle \phi \mid A \rangle$ is introduced by $\mathtt{mod}_\phi(M)$, where M is an element of A, subject to the restriction that M may only use variables with annotation ϕ^{n+1}. More concretely, when we construct M we (1) lose access to all id-annotated variables and (2) replace a variable $x : (\phi^{n+1} \mid A)$ with $x : (\phi^n \mid A)$. As only variables with identity annotation can be used with the variable rule, this means that within $\mathtt{mod}_\phi(-)$ we may use ϕ-annotated variables freely.

For instance, in the context with variables $x_0 : (\mathsf{id} \mid \mathtt{Nat})$, $x_1 : (\phi \mid \mathtt{Nat})$, and $x_2 : (\phi \circ \phi \mid \mathtt{Nat})$ the following programs are well-typed:

$$x_0 : \mathsf{Nat} \qquad \mathtt{mod}_\phi(x_1) : \langle \phi \mid \mathtt{Nat} \rangle \qquad \mathtt{mod}_\phi(\mathtt{mod}_\phi(x_2)) : \langle \phi \mid \langle \phi \mid \mathtt{Nat} \rangle \rangle$$

On the other hand, both $x_1 : \mathtt{Nat}$ and $\mathtt{mod}_\phi(x_0) : \langle \phi \mid \mathtt{Nat} \rangle$ are ill-typed as the annotations on variables do not match their usage.

This idea generalizes: to construct an element of $\langle \phi^k \mid A \rangle$ we use $\mathtt{mod}_{\phi^k}(M)$ where $M : A$ in a context where we have (1) lost access to variables with annotations ϕ^l where $l < k$ (2) replaced each variable $x : (\phi^{n+k} \mid A)$ with $x : (\phi^n \mid A)$. In the same context as the example above therefore, $\mathtt{mod}_{\phi\circ\phi}(x_2) : \langle \phi \circ \phi \mid \mathtt{Nat} \rangle$. We refer to the modification to the context given by (1) and (2) as ϕ^k*-restricting the context*.

Let us now consider the modal function type $(\mu \mid A) \to B$. An element of $(\mu \mid A) \to B$ is precisely a function which binds a variable of type A with annotation μ. Application for these function types $R(M)_\mu$ takes μ into account in the following way: $R(M)_\mu : B$ if (1) R has type $(\mu \mid A) \to B$ and (2) after μ-restricting the context, M has type A.

One feature remains to be discussed, the elimination principle for modal types:

$$\mathtt{let}_\nu\ \mathtt{mod}_\mu(y) \leftarrow R\ \mathtt{in}\ M\ \mathtt{over}\ x.C$$

To a first approximation, this principle allows us to replace a variable $x : (\nu \mid \langle \mu \mid A \rangle)$ with $y : (\nu \circ \mu \mid A)$. More precisely, $\mathtt{let}_\nu\ \mathtt{mod}_\mu(y) \leftarrow R\ \mathtt{in}\ M\ \mathtt{over}\ x.C : C[M/x]$ if (1) after binding $x : (\nu \mid \langle \mu \mid A \rangle)$, C is a type (2) after ν-restricting the context M has type $\langle \mu \mid A \rangle$ and (3) after binding $y : (\nu \circ \mu \mid A)$, R has type $C[\mathtt{mod}_\mu(y)/x]$.

Multiple modalities

The above approach for ϕ-restriction based on decrementing modal annotations provides a simple mental model for MTT. To extend these ideas to more complex mode theories, however, a more refined approach is necessary. We begin by discussing a small adjustment to the concepts introduced previously.

Rather than eagerly decrementing the annotation on a variable when we restrict a context, we instead *lazily* perform this update. Accordingly, we annotate each

variable with a pair of modalities and write $x :_{\mu/\nu} A$ for a μ-annotated variable with a ν-restriction lazily performed upon it. The rule for applying a restriction to a variable now becomes more uniform: to restrict $x :_{\mu/\nu} A$ by ξ we replace it with $x :_{\mu/\nu\circ\xi} A$. The variable rule applies only when the fraction 'cancels' i.e., $x :_{\mu/\mu} A \vdash x : A$.

For the mode theory under consideration, this is merely a change in notation as the behavior of the annotations of $x :_{\phi^l/\phi^k} A \vdash x : A$ is entirely determined by the difference $l - k$. We therefore introduce the following mode theory to illustrate the need for the 'lazy' approach:

Definition 4.2.2. Denote by $\mathscr{M}_1^{\mathsf{ex}}$ the mode theory with one mode and two generating modalities ψ and ϕ. The preorder is generated by the inequality $\psi \circ \psi \leq \phi$.

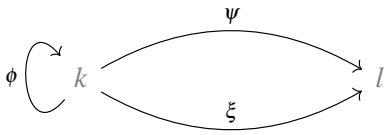
This mode theory introduces two new concepts simultaneously: multiple modalities and non-trivial inequalities between those modalities. Fortunately, to refine the idea explained above of μ-restricting a context, only one rule must be altered: To account for the preorder on modalities, we relax the variable rule slightly: $x :_{\mu/\nu} A \vdash x : A$ if $\mu \leq \nu$. With this modified rule, we can construct a coercion $\langle \psi \circ \psi \mid A \rangle \to \langle \phi \mid A \rangle$:

$$\mathtt{coerce} = \lambda x.\ \mathtt{let}_{\mathsf{id}}\ \mathtt{mod}_{\psi\circ\psi}(y) \leftarrow x\ \mathtt{in}\ \mathtt{mod}_{\phi}(y)\ \mathtt{over}\ \_.\ \langle \phi \mid A \rangle$$

Multiple modes and multiple modalities

Only one generalization is required at this point to provide a complete description of MTT: multiple modes. While thus far we have confined ourselves to discussing multiple modalities on one mode, we are allowed to have multiple modes in MTT as well. Consider the following mode theory:

Definition 4.2.3. $\mathscr{M}_2^{\mathsf{ex}}$ is the mode theory equipped with two modes k and l whose modalities are generated by $\phi : k \to k$ and $\psi, \xi : k \to l$. The preorder on hom-sets are generated by the inequalities $\mathsf{id}_k \leq \phi$ and $\xi \leq \psi$:



We note that now $\mathscr{M}_2^{\mathsf{ex}}$ now has two different modes k and l. Each mode in MTT gives rise to a separate type theory so that we must check not only that some term has a type, but also that the term, type, and all variables in scope live at the correct mode.

All of the standard constructions do not change the mode; thus, e.g., $\mathtt{succ}(n)$ will be well-typed at type `Nat` at mode m just when the same is true of n. We will notate "M has type A at mode m" by $M : A \mathbin{@} m$. Prior to discussing the two type constructors involving modalities, we must explain what it means for a context to be well-formed at mode m.

Definition 4.2.4. A variable declaration $x :_{\mu/\nu} A$ is well-formed at mode m if the following hold:

1. $\mu : n \longrightarrow o$ and $\nu : m \longrightarrow o$ for some o.

2. A is a type at mode n.

The context is well-formed at mode m if all variables in scope are well-formed at mode m.

Example 35. Restricting a well-formed context at m by $\mu : n \longrightarrow m$ yields a well-formed context in mode n.

It is worth emphasizing the contravariant nature of the restriction ν in $x :_{\mu/\nu} A$. This is crucial for the rules governing $\langle \mu \mid A \rangle$. The type $\langle \mu \mid A \rangle$ is well-formed at mode m if (1) $\mu : n \longrightarrow m$ for some n and (2) after μ-restricting the context, A is well-formed at mode n. In particular, $\langle \mu \mid - \rangle$ sends types at mode n to types at mode m so restriction must move contexts contravariantly from mode n to mode m. We remark, however, that aside from the additional checks to ensure that types are well-moded, this is the same rule as given previously. Likewise, the rules for introduction and elimination along with all of those for modal dependent products are merely instrumented with additional checks to ensure that types and terms live at the correct mode.

We conclude with a few examples.

Example 36. $\lambda x.x : (\xi \mid A) \to \langle \psi \mid A \rangle \ @ \ l$ is well typed. In particular, since $\xi \leq \psi$ we conclude $x :_{\xi/\psi} A \vdash x : A \ @ \ k$.

Example 37. We will define a function of the following type:

$$f : \langle \psi \mid \langle \phi \mid \mathtt{Nat} \rangle \rangle \to \langle \psi \circ \phi \mid \mathtt{Nat} \rangle \ @ \ l$$

We begin by binding a variable $x :_{\mathsf{id}/\mathsf{id}} \langle \psi \mid \langle \phi \mid \mathtt{Nat} \rangle \rangle$ so it now suffices to construct a term $\langle \psi \circ \phi \mid \mathtt{Nat} \rangle \ @ \ l$. To this end, we use the modal elimination principle on x to obtain a new variable $y :_{\psi/\mathsf{id}} \langle \phi \mid \mathtt{Nat} \rangle$. Applying modal elimination to y, we obtain $z :_{\psi\circ\phi/\mathsf{id}} \mathtt{Nat}$.

We still wish to construct a term $\langle \psi \circ \phi \mid \mathtt{Nat} \rangle$. Applying the modal introduction rule, we $\psi \circ \phi$ restrict the context (so y becomes $y :_{\psi\circ\phi/\psi\circ\phi} \mathtt{Nat}$). Our goal is then $\mathtt{Nat}$, so y suffices.

All told, the term final term is as follows:

$$\begin{array}{l}
\lambda x. \\
\ \mathtt{let}_{\mathsf{id}_k}\ \mathtt{mod}_{\psi}(y) = x\ \mathtt{in} \\
\ \ \mathtt{let}_{\psi}\ \mathtt{mod}_{\phi}(z) = y\ \mathtt{in} \\
\ \ \ \mathtt{mod}_{\psi\circ\phi}(z) \\
\ \ \mathtt{over}\ \langle \psi \circ \phi \mid \mathtt{Nat} \rangle \\
\ \mathtt{over}\ \langle \psi \circ \phi \mid \mathtt{Nat} \rangle
\end{array}$$

4.3 Normalization by Evaluation

A crucial ingredient of any type checker is a procedure for determining when two types are equal. In `mitten`, we have implemented this decision procedure through a normalization algorithm: a function which sends a term to a corresponding *normal form*. The precise definition of normal form is then less important than the fact that definitional equality for normal forms is straightforward to decide. Writing $\mathsf{NfTerms}$ for the collection of normal forms, we view our normalization algorithm as a function:

$$\underline{\mathbf{norm}}_\Gamma : \mathsf{Syntax} \to \mathsf{NfTerms}$$

Merely having a function from syntax to normal forms, however, is insufficient to decide definitional equality. Accordingly, we are interested in normalization functions which satisfy the following properties:

Definition 4.3.1. A normalization function is called *complete* if $\Gamma \vdash A = B \mathbin{@} m$ implies $\underline{\mathbf{norm}}_\Gamma(A) = \underline{\mathbf{norm}}_\Gamma(B)$

Definition 4.3.2. A normalization function is *sound* if $\Gamma \vdash A \mathbin{@} m$ implies $\Gamma \vdash \underline{\mathbf{norm}}_\Gamma(A) = A \mathbin{@} m$.

Completeness states that normalization lifts to a function on syntax quotiented by definitional equality while soundness states that this induced function has a section. Taken together, therefore, we have the following:

Corollary 4.3.3. *Let* $\underline{\mathbf{norm}}_\Gamma$ *be sound and complete then* $\Gamma \vdash A = B \mathbin{@} m$ *if and only if* $\underline{\mathbf{norm}}_\Gamma(A) = \underline{\mathbf{norm}}_\Gamma(B)$.

The traditional approach to constructing a normalization function is to specify an untyped rewriting system which directs and presents the equational theory. Equality of terms is then convertibility within this rewriting system so that strong normalization ensures both soundness and completeness. This approach, however, turns out to be unworkable for more elaborate dependent type theories with type-directed rules. One possible approach is to can refine a rewriting system to be type-directed system which—in conjunction with other mechanisms—can decide conversion directly [5], we adopt an entirely different approach to associating terms to normal forms: *normalization by evaluation* (NbE).

Normalization by evaluation breaks the process of normalizing a term into two distinct phases: evaluation and quotation. The first *evaluates* a term into a *semantic domain*. For our purposes, the semantic domain is simply a more restrictive form of syntax which disallows β-reducible terms. The process of evaluation boils down to placing a term in β-normal form while crucially retaining various pieces of type information for the next phase. The second phase, quotation, takes an element of the semantic domain and *quotes* it back to syntax. In the process it η expands terms wherever possible. As a result, the full loop of evaluation and quotation sends a term to its β-normal η-long form as required. Figure 4.1 gives a graphical overview of the process.

We describe the semantic domain in detail in Section 4.3. The actual algorithm is described over the following three sections (Section 4.3). Our algorithm is inspired by Gratzer's gluing-based argument for normalization [54] and we conjecture that this link can be made sufficiently precise to establish the soundness and completeness of our code.

Remark 38. The version of normalization-by-evaluation we use is robust enough to require only local modifications in order to accommodate modal types. Accordingly, we focus primarily on connectives like dependent products and modal types whose behavior is impacted and refer the reader to, e.g., Abel [3] for a description how the algorithm works on the remaining connectives.

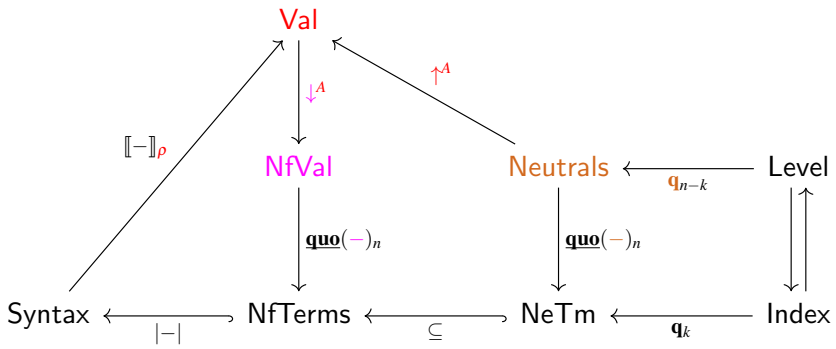


Figure 4.1: Overview of the algorithm inspired by [57] and [3].

The Domain

We start by a brief overview of the semantic domains described in Fig. 4.1:

$$
\begin{array}{llll}
\textit{(values)} & A,u & ::= & \uparrow^A e \mid \lambda(f) \mid (\mu \mid A) \to B \mid \mathsf{zero} \mid \mathsf{suc}(v) \mid \mathsf{Nat} \\
 & & & \mid (v_1, v_2) \mid A \times B \mid \langle \mu \mid A \rangle \mid \mathsf{mod}_\mu(v) \\
\textit{(neutrals)} & e & ::= & \mathbf{q}_k \mid \mathsf{app}[\mu](e,d) \mid \mathsf{pr}_1(e) \mid \mathsf{pr}_2(e) \\
 & & & \mid \mathsf{letmod}(\mu,\nu,C,c,A,e) \mid \mathsf{rec}(C,u,v,e) \\
\textit{(environments)} & \rho & ::= & \cdot \mid \rho.v \\
\textit{(closures)} & C,f & ::= & \mathsf{clo}(M,\rho) \\
\textit{(normals)} & d & ::= & \downarrow^A v
\end{array}
$$

Informally, *neutral forms* are generated by variables and elimination forms stuck on other neutrals. To a first approximation, a neutral is a chain of eliminations which are stuck on a variable. On the other hand, *values*—the codomain of the evaluation function—are primarily generated by introduction forms. In particular, there are no elimination forms directly available on values and there is no uniform way to turn a value into a neutral form. Consequently, β-reducible terms cannot be expressed in this grammar. One can, however, lift a neutral into a value after annotating the neutral form with its type. Tersely, values are β-short but not necessarily η-long.

A defining aspect of our approach to NbE is the handling of open terms. Rather than directly evaluating under a binder, when we reach, e.g., a lambda, we suspend the computation and store the intermediate result in a *closure*. The evaluation is resumed as soon as further information is gathered. In the case of a function, for instance, the evaluation of the body is resumed only after the function is applied. A closure is a combination of the term being evaluated and "the state of the evaluation algorithm." The latter amounts to the environment of variables which is reified and stored in the closure alongside the term.

Normal forms have only one constructor, *reification*. Values are lifted to normals by annotating them with a type. This type annotation is used during the quotation process in Section 4.3 in order to deal with the η-laws.

We emphasize that while terms use De Bruijn indices, neutral forms use De Bruijn *levels* to represent variables. This small maneuver ensures that values, neutral forms, and normal forms are silently weakened and we will capitalize on this fact throughout our algorithm [3].

Evaluation

Evaluation is the procedure of interpreting syntax into the semantic domains, specifically values. At a high-level, this amounts to β-reducing all terms (recall β-reducible terms cannot be represented as values). The presence of variables, however, causes some elimination forms to become stuck. These stuck terms are evaluated into neutrals and annotated with a type.

We single out a few interesting cases of the evaluation algorithm shown in Figure 4.2.

$$\boxed{[\![\_]\!]_{\_} : \mathsf{Syntax} \to \mathsf{Env} \to \mathsf{Val}}$$

$$\textsc{eval/var}\ \frac{\rho(i) = v}{[\![\mathbf{q}_i]\!]_\rho = v} \qquad \textsc{eval/pi}\ \frac{[\![A]\!]_\rho = A_0}{[\![(\mu \mid A) \to B]\!]_\rho = (\mu \mid A_0) \to \mathsf{clo}(B, \rho)} \qquad \textsc{eval/modify}\ \frac{[\![A]\!]_\rho = A_0}{[\![\langle \mu \mid A \rangle]\!]_\rho = \langle \mu \mid A_0 \rangle}$$

$$\textsc{eval/mod}\ \frac{[\![M]\!]_\rho = v}{[\![\mathsf{mod}_\mu(M)]\!]_\rho = \mathsf{mod}_\mu(v)} \qquad \textsc{eval/app}\ \frac{[\![M]\!]_\rho = u \qquad [\![N]\!]_\rho = v}{[\![M(N)_\mu]\!]_\rho = \underline{\mathbf{app}}(u, v)}$$

$$\textsc{eval/letmod}\ \frac{[\![M]\!]_\rho = v}{[\![\mathsf{let}_v\ \mathsf{mod}_\mu(\_) \leftarrow M\ \mathsf{in}\ N : A]\!]_\rho = \underline{\mathbf{letmod}}_{v;\mu}(\mathsf{clo}(A, \rho), \mathsf{clo}(N, \rho), v)}$$

$$(\rho.v)(0) = v \qquad\qquad (\rho.v)(i+1) = \rho(i)$$

Figure 4.2: Evaluation function, selected cases.

The work of evaluation is done around eliminators. Therefore, we single these cases out and define 'helper' functions for this portion of the algorithm. The interesting new cases are **letmod** and **app**, but generally for every syntax elimination form we define a suggestively named function that automatically beta-reduces eliminators applied to an introduction form, or returns a neutral and annotates it with its type.

$$\boxed{\underline{\mathbf{app}}(u,v) : \mathsf{Val} \qquad \underline{\mathbf{proj}}_i(v) : \mathsf{Val} \qquad \underline{\mathbf{letmod}}_{\nu;\mu}(C,c,v) : \mathsf{Val} \qquad \underline{\mathbf{J}}(C,c_{\mathsf{refl}},p) : \mathsf{Val}}$$

$$\frac{}{\underline{\mathbf{inst}}(\mathsf{clo}(M,\rho),v) = [\![M]\!]_{\rho.v}} \qquad \frac{C = \lambda(C)}{\underline{\mathbf{app}}(u,v) = \underline{\mathbf{inst}}(C,v)}$$

$$\frac{u = \uparrow^{A_0} e \qquad A_0 = (\mu \mid A) \to C \qquad \underline{\mathbf{inst}}(C,v) = B}{\underline{\mathbf{app}}(u,v) = \uparrow^{B} \mathsf{app}[\mu](e, \downarrow^{A} v)}$$

$$\frac{v = \mathsf{mod}_{\mu}(v') \qquad \underline{\mathbf{inst}}(c,v') = u}{\underline{\mathbf{letmod}}_{\nu;\mu}(C,c,v) = u}$$

$$\frac{v = \uparrow^{A_0} e \qquad A_0 = \langle \mu \mid A \rangle \qquad \underline{\mathbf{inst}}(C, \uparrow^{\langle \mu | A \rangle} e) = B}{\underline{\mathbf{letmod}}_{\nu;\mu}(C,c,v) = \uparrow^{B} \mathsf{letmod}(\nu,\mu,C,c,A,e)}$$

As mentioned previously, we use closures to represent syntax that cannot be evaluated in the present environment. Once we have found the value to complete the environment, we *instantiate* the closure with it and continue the evaluation in the extended environment.

Quotation

Quotation is the process of turning normals into terms. We will ensure that the results of quotation are always *normal form terms*, that is, β-short and η-long terms.

To account for the fact that normal forms mention values and neutral forms, quotation is split into three mutually recursive functions. Quotation must perform η-expansion and is therefore type-directed. Accordingly, while we have a quotation procedure which applies to values, this portion of the algorithm can only be used for quoting types where there is no associated η-expansions. All three of these functions take a natural number in addition to the actual term being quoted. This number represents the next available De Bruijn level for a free variable; it is used to quote terms with binders.

We present the novel cases of quotation of normal forms—those with modalities—below:

$$\frac{A_0 = (\mu \mid A) \to B \qquad \underline{\mathbf{inst}}(B, \uparrow^{A} \mathbf{q}_k) = B \qquad \underline{\mathbf{quo}}(\downarrow^{B} \underline{\mathbf{app}}(v, \uparrow^{A} \mathbf{q}_k))_{k+1} = M}{\underline{\mathbf{quo}}(\downarrow^{A_0} v)_k = \lambda(M)}$$

$$\frac{A_0 = \langle \mu \mid A \rangle \qquad v = \mathsf{mod}_\mu(w)}{\underline{\mathbf{quo}}(\downarrow^{A_0} v)_k = \mathsf{mod}_\mu(\underline{\mathbf{quo}}(\downarrow^{A} w)_k)} \qquad \frac{A_0 = \langle \mu \mid A \rangle \qquad v = \uparrow^{B} e}{\underline{\mathbf{quo}}(\downarrow^{A_0} v)_k = \underline{\mathbf{quo}}(e)_k}$$

$$\frac{A_0 = \uparrow^{A} e \qquad v = \uparrow^{A'} e}{\underline{\mathbf{quo}}(\downarrow^{A_0} v)_k = \underline{\mathbf{quo}}(e)_k}$$

We draw attention to one aspect of the first rule. This rule quotes a function, so consider the case where $v = \lambda(\mathsf{clo}(M, \rho))$. We create a fresh variable $\uparrow^{A} \mathbf{q}_k$ and make the semantic application $\underline{\mathbf{app}}(\lambda(\mathsf{clo}(M,\rho)), \uparrow^{A} \mathbf{q}_k)$. This last step is only sensible because values are closed under silent weakening; otherwise ρ would need to be weakened over $\mathbf{q}_k$.

Finally, we record the novel cases of quotation for neutral forms:

$$\underline{\mathbf{quo}}(\mathsf{app}[\mu](e,d))_k = \underline{\mathbf{quo}}(e)_k(\underline{\mathbf{quo}}(d)_k)_\mu$$

$$\frac{\underline{\mathbf{inst}}(C, \mathsf{mod}_\mu(\uparrow^{A} \mathbf{q}_k)) = B \qquad \underline{\mathbf{inst}}(c, \uparrow^{A} \mathbf{q}_k) = v}{\underline{\mathbf{quo}}(\mathsf{letmod}(\nu, \mu, C, c, A, e))_k = \mathsf{let}_\nu\ \mathsf{mod}_\mu(\_) \leftarrow \underline{\mathbf{quo}}(e)_k\ \mathsf{in}\ \underline{\mathbf{quo}}(\downarrow^{B} v)_{k+1}}$$

The NbE function

Having defined both evaluation and quotation, we are almost in a position to define the complete normalization algorithm. The only missing step is the construction of the *initial environment* from a context. This portion of the algorithm takes a context Γ and produces an environment consisting of the variables bound in Γ. We then use this environment to kick off the evaluation of terms in context Γ:

$$\underline{\mathbf{reflect}}(\mathbf{1}) = \cdot \qquad \underline{\mathbf{reflect}}(\Gamma.(\mu \mid A)) = \underline{\mathbf{reflect}}(\Gamma).\uparrow^{[\![A]\!]_{\underline{\mathbf{reflect}}(\Gamma)}} \mathbf{q}_{|\Gamma|}$$

Finally, the complete normalization algorithm evaluates a term $\Gamma \vdash M : A \mathbin{@} m$ in the initial environment specified by Γ and quotes it back:

$$\underline{\mathbf{norm}}_{\Gamma,A}(M) = \underline{\mathbf{quo}}(\downarrow^{[\![A]\!]_{\underline{\mathbf{reflect}}(\Gamma)}} [\![M]\!]_{\underline{\mathbf{reflect}}(\Gamma)})_{|\Gamma|}$$

4.4 Implementing a Mode Theory

Thus far we have been somewhat vague about which mode theory we were instantiating MTT with. The normalization algorithm given in Section 4.3, for instance, did not need to manipulate or compare modalities and so this point was easy to gloss over. The type-checker, on the other hand, must manipulate and scrutinize modalities and its definition requires a precise specification of a mode theory. Accordingly, we now present a representation of mode theories and operations upon them suitable for implementing a type-checker.

Concretely, our presentation follows the actual representation of mode theories used in `mitten`, our implementation of MTT. In `mitten`, all information specific to a

```
type mode
val eq_mode : mode → mode → bool
```

```
type m
val idm : m
val compm : m → m → m
val dom_mod : m → mode → mode
val cod_mod : m → mode → mode
val (=) : m → m → bool
val (≤) : m → m → bool
```

Figure 4.3: A fragment of the signature for mode theories used in `mitten`.

mode theory is confined to a single OCaml module on which the type-checker depends. In particular, to configure `mitten` with a new mode theory, it is only necessary to implement that single module. There are three parts to our signature for mode theories (summarized in Fig. 4.3):

1. Two abstract types; one for modes and one for modalities.

2. Various operations to compose modalities, extract the domain or codomain mode from a modality, or construct the identity modality.

3. Three operations to compare modes for equality and modalities for (in)equality.

It is these last two operations which are particularly crucial. Recall that not all mode theories admit decidable (in)equality and without it, type-checking MTT is undecidable. Accordingly, any implementation of MTT will require the user to supply a decision procedure for the mode theory. Our implementation shows that this information is both necessary and essentially sufficient. We note that the decision procedures for mode theories are completely separate from the terms and types of MTT and no knowledge of e.g., normalization-by-evaluation is required for their implementation.[2]

Remark 39. The reader might wonder why `idm` is not parametrized over `mode`. This is because `idm` internally is a placeholder for some identity modality, whose mode is elaborated. This alleviates practitioners of some tedious bookkeeping obligations in their proofs. This approach necessitates that the boundary projections `dom_mod` and `cod_mod` take an additional argument of type `mode`, which is returned on input `idm`. Essentially, we assume that always one part of the boundary is known so `dom_mod` gets a modality and its codomain as argument whereas `cod_mod` gets a modality and its domain as argument.

[2]See the following for examples: https://github.com/logsem/mitten_preorder/blob/main/src/lib/

4.5 Semantic Type-Checking Algorithm

Having defined the normalization algorithm, we now define the type-checking algorithm for MTT. As mentioned in Sections 4.1 and 4.2, the algorithm is a variant of Coquand's semantic type checking algorithm for dependent type theory [40]. Accordingly, the algorithm breaks into two distinct phases: checking and synthesis. The checking portion of the algorithm accepts a context Γ, a term M, and a type A and checks that M has type A in context Γ. The synthesis phase accepts only the context and term, and synthesizes the type of the term in this context.

This simple picture is slightly complicated in the case of MTT, where various side conditions must be managed. For instance, we must ensure that the modalities a user writes in modal types are well-formed and that the term and type exist at the same mode as the context. These same considerations also require us to form a more intricate notion of a *semantic context* specifically for the type-checking algorithm.

We discuss the definition of semantic contexts in Section 4.5 and present a representative fragment of the type-checking algorithm itself in Section 4.5.

Semantic Contexts

In Section 4.2, we explained the intuitions behind MTT while working informally with the collection of variables in scope. Prior to discussing the type checker, we must describe the precise notion of context to organize these variables. Two factors complicate this otherwise standard structures: the modal annotations and restrictions and the need to evaluate terms during type-checking.

To a first approximation, contexts are still lists of variables with types but with additional bells and whistles added in order to support these two requirements. In order to record the necessary modal information, each variable is annotated by a modality. Deviating from Section 4.2, we add a new context operation $\Xi.\{\mu\}$ to 'lazily' restrict all entries in a context Ξ by μ rather than storing this information on each variable separately.

For the second requirement, recall that type-checking must repeatedly test when two types are equal for the *conversion rule*. Accordingly, the context must store enough information to support this conversion test. We follow Coquand [40] and represent each type in the context by the corresponding *value* (in the sense of Section 4.3) and pair each variable with a corresponding value. This value may just be $\uparrow^A \mathbf{q}_i$, but it may also store the term associated definition. By storing information in this form, we can easily project out a *semantic environment* of a context and use that to evaluate a term and check for convertibility during type checking.

The grammar of *semantic contexts* is presented below:

$$\textit{(semantic contexts)} \quad \Xi \quad ::= \quad \cdot \mid \Xi.(v :_\mu A \mathbin{@} m) \mid \Xi.\{\mu\}$$

We now define two functions: The partial *lookup function*, which displays the type with its annotation and restriction as well as the mode it lives at, and the *stripping* function, which returns an environment by projecting out only the value components

of the semantic context. The lookup function is undefined whenever a De Bruijn index is larger than the length of the context.

$$
\begin{aligned}
&(\Xi.(v :_\mu A \mathbin{@} m))(0) = (\mu|A)_m, \{\mathsf{id}\} \\
&(\Xi.(w :_\xi B \mathbin{@} o))(i+1) = (\mu|A)_m, \{v\} && \text{where } (\mu|A)_m, v = \Xi(i) \\
&(\Xi.\{\nu'\})(i) = (\mu|A)_m, \{v \circ \nu'\} && \text{where } (\mu|A)_m, v = \Xi(i) \\
&|\cdot| = \cdot \\
&|\Xi.(v :_\mu A \mathbin{@} m)| = |\Xi|.v \\
&|\Xi.\{\mu\}| = |\Xi|
\end{aligned}
$$

Notation 4.5.1. If we extend a semantic context with a type where the value is a fresh variable, we hide it to make the expression more readable. $||\Xi||$ denotes the De Bruijn level.

$$\Xi.(\mu|A) \triangleq \Xi.(\uparrow^A \mathbf{q}_{||\Xi||} :_\mu A \mathbin{@} m)$$

If the modality μ is furthermore the identity modality, we omit it and write

$$\Xi.A \triangleq \Xi.(\uparrow^A \mathbf{q}_{||\Xi||} :_{\mathsf{id}} A \mathbin{@} m)$$

Checking and Synthesis

We now come to the type-checking algorithm which is split into a pair of judgments: $\Xi \vdash M \Leftarrow A \mathbin{@} m$ and $\Xi \vdash R \Rightarrow A \mathbin{@} m$. The first, $\Xi \vdash M \Leftarrow A \mathbin{@} m$, handles type checking which tests if M has type A in Ξ. The second, $\Xi \vdash M \Rightarrow A \mathbin{@} m$, implements type synthesis and accordingly takes only the semantic context Ξ and term M and returns type A of M in context Ξ if one can be inferred.

We present a few representative rules for these judgments (and explain them below). To ensure that terms and types are well-formed, we utilize the functions exposed by the signature presented in Section 4.4. In particular, $n \stackrel{?}{=} m$ checks whether two modes are equal and $\mu \leq \nu$ is the modality ordering relation. Furthermore, with $\mu.\mathsf{dom}$ and $\mu.\mathsf{cod}$ we denote the respective domain and codomain of a modality—denoted `dom_mod` and `cod_mod` respectively in Section 4.4. For readability, we leave the second argument of $\mu.\mathsf{dom}$ and $\mu.\mathsf{cod}$ implicit.

PI

$$\frac{\Xi.\{\mu\} \vdash A \Leftarrow \mathsf{U} \mathbin{@} \mu.\mathsf{dom} \qquad \Xi.(\mu|A) \vdash B \Leftarrow \mathsf{U} \mathbin{@} m \qquad \mu.\mathsf{cod} \stackrel{?}{=} m}{\Xi \vdash (\mu \mid A) \to B \Leftarrow \mathsf{U} \mathbin{@} m}$$

MOD-FORM

$$\frac{\Xi.\{\mu\} \vdash A \Leftarrow \mathsf{U} \mathbin{@} \mu.\mathsf{dom} \qquad \mu.\mathsf{cod} \stackrel{?}{=} m}{\Xi \vdash \langle \mu \mid A \rangle \Leftarrow \mathsf{U} \mathbin{@} m}$$

MOD-INTRO

$$\frac{\Xi.\{\mu\} \vdash M \Leftarrow A \mathbin{@} \mu.\mathsf{dom} \qquad \mu.\mathsf{cod} \stackrel{?}{=} m}{\Xi \vdash \mathtt{mod}_\mu(M) \Leftarrow \langle \mu \mid A \rangle \mathbin{@} m}$$

CONV

$$\frac{\Xi \vdash R \Rightarrow B \mathbin{@} m \qquad A \equiv_{|\Xi|} B}{\Xi \vdash R \Leftarrow A \mathbin{@} m}$$

$$\textsc{var}\ \frac{\Xi(k) = (\mu|A)_m, \nu \qquad \mu \leq \nu \qquad m \stackrel{?}{=} n}{\Xi \vdash \mathbf{q}_k \Rightarrow A \ @\ n}$$

$$\textsc{mod-elim}\ \frac{\nu.\mathsf{cod} \stackrel{?}{=} m \qquad \Xi.\{\nu\} \vdash R \Rightarrow \langle \mu \mid A \rangle \ @\ \nu.\mathsf{dom} \qquad \Xi.(\nu, \langle \mu \mid A \rangle) \vdash C \Leftarrow \mathsf{U} \ @\ m \qquad \Xi.(\nu \circ \mu, A) \vdash N \Leftarrow [\![C]\!]_{|\Xi|.\mathsf{mod}_\mu(\uparrow^A \mathbf{q}_{||(||\Xi)})} \ @\ m}{\Xi \vdash \mathtt{let}_\nu\ \mathtt{mod}_\mu(\_) \leftarrow R\ \mathtt{in}\ N\ \mathtt{over}\ C \Rightarrow [\![C]\!]_{|\Xi|.[\![R]\!]_{|\Xi|}} \ @\ m}$$

We first consider the formation rule for dependent products. First we verify that indeed $\mu.\mathsf{cod} \stackrel{?}{=} m$ to ensure that the modality μ can be used at this mode. Recall that Π-types in MTT go from a μ-restricted type A to a non restricted type B. Accordingly, we check that A is a type in the μ-restricted semantic context $\Xi.\{\mu\}$ and that B is well-formed in the context $\Xi.(\mu|A)$. Note that when checking A we change the mode to $\mu.\mathsf{dom}$.

Since the modal formation and introduction rules follow a similar pattern we will only look at the modal introduction rule. To validate that $\mathtt{mod}_\mu(M)$ has type $\langle \mu \mid A \rangle$ at mode m we first verify that $\mu.\mathsf{cod} \stackrel{?}{=} m$. Then we check that M has type A in the μ-restricted environment $\Xi.\{\mu\}$ at mode $\mu.\mathsf{dom}$.

Next, we discuss the conversion rule. When considering a synthesizable term R, the type-checking algorithm proceeds somewhat differently. We first synthesize the type of R and then compare the result to the type we were given to check R against. It is this comparison which uses the normalization algorithm of Section 4.3 to compute the normal forms of A and B and decides afterwards the equality of the normalized expressions.

To synthesize a variable $\mathbf{q}_k$ in a semantic context Ξ at mode m we first compute the type of the variable together with its annotation and restriction $(\mu|A)_m, \{\nu\}$, using the lookup function defined in Section 4.5. Before we return A as the type of $\mathbf{q}_k$, we must also perform an additional check to ensure that $\mu \leq \nu$ so that this occurrence of the variable is valid.

Finally, we consider the modal elimination case. Recall from Section 4.2 that the modal elimination principle allows us to 'pattern-match' on a term $R : \langle \mu \mid A \rangle$ in a ν-restricted context and replace it with a variable $x :_{\nu \circ \mu} A$. To synthesize $\mathtt{let}_\nu\ \mathtt{mod}_\mu(\_) \leftarrow R\ \mathtt{in}\ N\ \mathtt{over}\ C$, we take advantage of the fact that the user provides the motive C already; if this term is well-typed, its type must be $[\![C]\!]_{|\Xi|.[\![R]\!]_{|\Xi|}}$.

There are, however, several checks to perform to ensure that the term is actually well-typed. First, we check that $\nu.\mathsf{cod} \stackrel{?}{=} m$. Next, we synthesize the type of R in the ν-restricted context and check that the result is of the form $\langle \mu \mid A \rangle$. Having computed $\langle \mu \mid A \rangle$, we then check that both C and N are well-formed. The motive C must be a type in the extended context $\Xi.(\nu, \langle \mu \mid A \rangle)$ while N must have type $[\![C]\!]_{|\Xi|.[\![R]\!]_{|\Xi|}}$ in context $\Xi.(\nu, \langle \mu \mid A \rangle)$.

A complete implementation of the algorithm can be found at https://github.com/logsem/mitten_preorder/blob/main/src/lib/check.ml.

4.6 Case study: guarded recursion in `mitten`

We now discuss an extended example using `mitten` with a particular choice of mode theory. By instantiating `mitten` appropriately, we convert it into a proof assistant for *guarded recursion* and use it to reason about classical examples from the theory.

Guarded recursion

Guarded recursion provides a discipline for managing recursive definitions within type theory without compromising soundness. In particular, guarded type theory extends type theory with a handful of modalities ($\blacktriangleright$, Γ and Δ) along with a modified version of the fixed-point combinator:

$$\mathsf{loeb} : (\blacktriangleright A \to A) \to A$$

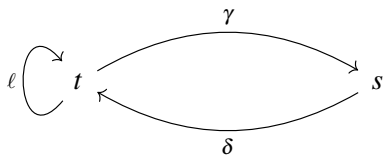
By placing the recursive call under a $\blacktriangleright$, this weakened fixed-point combinator does not result in inconsistencies. Together with the other modalities, moreover, it can be used to define and reason about coinductive types and gives rise to a *synthetic* form of domain theory.

Following [23], we are interested in using `mitten` as a tool to reason about a particular *model* of guarded recursion: $\mathbf{PSh}(\omega)$. In fact, using MTT's capacity to reason about multiple categories at once, we will work with a slightly richer model which includes both $\mathbf{PSh}(\omega)$ and $\mathbf{Set}$. In this model, the aforementioned modalities are all interpreted by right adjoints:

$$\Gamma : \mathbf{PSh}(\omega) \to \mathbf{Set} \qquad \Delta : \mathbf{Set} \to \mathbf{PSh}(\omega) \qquad \blacktriangleright : \mathbf{PSh}(\omega) \to \mathbf{PSh}(\omega)$$
$$\Gamma(X) = [\mathbf{1}, X] \qquad \Delta(S) = \lambda\_.\, S \qquad \blacktriangleright(X)(0) = \{\star\} \quad \blacktriangleright(X)(n+1) = X(n)$$

In particular, the composite of Γ and Δ is the global sections comonad $\Box$. The fixed-point operator loeb in $\mathbf{PSh}(\omega)$ is definable using induction over ω.

Gratzer et. al [58] have shown that MTT with a mode theory axiomatizing these three modalities is modeled by these two categories and therefore provides a suitable basis for guarded recursion. We recall their mode theory in Fig. 4.4.



$$\delta \circ \gamma \leq 1 \qquad 1 = \gamma \circ \delta$$
$$1 \leq \ell \qquad \gamma = \gamma \circ \ell$$
$$\mu \leq \nu \wedge \nu \leq \mu \implies \mu = \nu$$

Figure 4.4: $\mathscr{G}$: a mode theory for guarded recursion

The equalities represented in Fig. 4.4 together with the equational theory of MTT ensure that $\Box = \delta \circ \gamma$ is an idempotent comonad and that the following equivalence is definable:

$$\langle \Box \mid \langle \ell \mid A \rangle \rangle \simeq \langle \Box \mid A \rangle.$$

In order to actually reason about guarded definitions, however, we still must add Löb induction to the system. Adding Löb induction primitively raises substantial issues [56], so we opt to axiomitize it along with a (propositional) equation specifying its unfolding principle:

$$\mathsf{loeb} : ((\ell \mid A) \to A) \to A \mathbin{@} t$$

$$\mathsf{unfold} : (f : (\ell \mid A) \to A) \to \mathsf{Id}_A(\mathsf{loeb}\, f, f(\mathsf{loeb}\, f)) \mathbin{@} t$$

As to be expected, these new constants disrupt canonicity but crucially cause no issues for type checking. We now discuss how to instantiate `mitten` with this particular mode theory.

Implementation

In order to use `mitten` to reason about guarded MTT, we must construct an implementation of the mode theory module corresponding to Fig. 4.4 and extend `mitten` with constants for Löb induction. The latter point is routine; `mitten` supports adding axioms to a development. We therefore focus on the first step: the implementation of the mode theory.

The main challenge when implementing Fig. 4.4 is to show that the relation $\leq$ is decidable. We have done so by using a (simple) form of normalization-by-evaluation to reduce modalities in this mode theory to normal forms which can be directly compared.

Remark 40. We leave the modes during the evaluation implicit and assume, without loss of generality, that we are only considering well-formed modalities.[3]

By studying the category generated by Fig. 4.4, it becomes clear that $\mathscr{G}$ is far from a free mode theory. In fact, many possible compositions trivialize; in a chain of composable modalities we can freely remove any $\gamma \circ \delta$ as well as any ℓ to the right of a γ. Accordingly, there are only four kinds of expressions remaining which thus constitute *normal modalities*:

$$\textit{(Normal modalities)} \quad \mu, \nu \quad ::= \quad \ell^k \mid \ell^k \circ \delta \mid \ell^k \circ \delta \circ \gamma \mid \gamma \mid \mathsf{id}_s$$

Note that $k = 0$ is allowed and thus in particular $\delta \circ \gamma = \ell^0 \circ \delta \circ \gamma$ as well as $\mathsf{id} = \ell^0$. There is an evident map i sending a normal form μ to a modality in $\mathscr{G}$. We now construct an inverse to this map:

$$\begin{array}{ll}
\underline{\mathbf{eval}}(\mathsf{id}_t) = \ell^0 & \underline{\mathbf{comp}}(\ell, \ell^k) = \ell^{k+1} \\
\underline{\mathbf{eval}}(\mathsf{id}_s) = \mathsf{id}_s & \underline{\mathbf{comp}}(\ell, \ell^k \circ \delta) = \ell^{k+1} \circ \delta \\
\underline{\mathbf{eval}}(\ell \circ \nu) = \underline{\mathbf{comp}}(\ell, \underline{\mathbf{eval}}(\nu)) & \underline{\mathbf{comp}}(\ell, \ell^k \circ \delta \circ \gamma) = \ell^{k+1} \circ \delta \circ \gamma \\
\underline{\mathbf{eval}}(\gamma \circ \nu) = \underline{\mathbf{comp}}(\gamma, \underline{\mathbf{eval}}(\nu)) & \underline{\mathbf{comp}}(\gamma, \ell^k) = \gamma \\
\underline{\mathbf{eval}}(\delta \circ \nu) = \underline{\mathbf{comp}}(\delta, \underline{\mathbf{eval}}(\nu)) & \underline{\mathbf{comp}}(\gamma, \ell^k \circ \delta \circ \gamma) = \gamma
\end{array}$$

[3] This assumption is justified since `mitten` checks all modalities prior to normalization and type-checking.

$$\underline{\mathbf{comp}}(\gamma, \ell^k \circ \delta) = \mathsf{id}_s \qquad \underline{\mathbf{comp}}(\delta, \gamma) = \ell^0 \circ \delta \circ \gamma$$
$$\underline{\mathbf{comp}}(\delta, \mathsf{id}_s) = \ell^0 \circ \delta$$

Theorem 4.6.1. *For any modality μ we have that $\mu = i(\underline{\mathbf{eval}}(\mu))$.*

Next, we define a (decidable)partial ordering on normal modalities:

$$\frac{m \leq n}{\ell^m \sqsubseteq \ell^n} \qquad \frac{}{\gamma \sqsubseteq \gamma} \qquad \frac{m \leq n}{\ell^m \circ \delta \circ \gamma \sqsubseteq \ell^n \circ \delta \circ \gamma} \qquad \frac{m \leq n}{\ell^m \circ \delta \circ \gamma \sqsubseteq \ell^n}$$

$$\frac{m \leq n}{\ell^m \circ \delta \sqsubseteq \ell^n \circ \delta} \qquad \frac{}{\mathsf{id}_s \sqsubseteq \mathsf{id}_s}$$

Theorem 4.6.2. *For any normal modalities μ and ν we have $\mu \sqsubseteq \nu$ if and only if $i(\mu) \leq i(\nu)$.*

Corollary 4.6.3. *Equality of modes and inequality of modalities are both decidable.*

Streams in guarded `mitten`

We now illustrate the use of this instantiation of `mitten` by defining the types of guarded and coinductive streams and constructing various examples.

Remark 41. In the following we deviate from our surface syntax to enhance readibility of the derivations. Thus, we leave many arguments implicit and alter certain notations. In particular, propositional identites are denoted by $a \equiv b$ instead of $\mathsf{Id}_A(a,b)$ and implicit arguments are omitted. We furthermore hide the type family C of the modal elimination rule in the following constructions.

We begin with the type of *guarded streams*.

$$\mathsf{gstream\_fun} : \mathsf{U} \to (\ell \mid \mathsf{U}) \to \mathsf{U} \,@\, t \qquad \mathsf{gstream} : \mathsf{U} \to \mathsf{U} \,@\, t$$
$$\mathsf{gstream\_fun}\, A\, X = A \times \langle \ell \mid X \rangle \qquad \mathsf{gstream}\, A = \mathsf{loeb}(\mathsf{gstream\_fun}\, A)$$

Notation 4.6.4. We will make use of several standard functions for intensional identity types such as the functions $\mathsf{transport} : A \equiv B \to A \to B$ and $-^{-1} : a \equiv b \to b \equiv a$.

Recall that we have added Löb induction only with a *propositional* unfolding rule. Accordingly, we must use transport along this equality to obtain the folding and unfolding operations for gstream:

$$\mathsf{gfold} : (A : \mathsf{U}) \to A \times \langle \ell \mid \mathsf{gstream}\, A \rangle \to \mathsf{gstream}\, A \,@\, t$$
$$\mathsf{gfold}\, A = \mathsf{transport}\,(\mathsf{unfold}(\mathsf{gstream\_fun}\, A))^{-1}$$

$$\mathsf{gunfold} : (A : \mathsf{U}) \to \mathsf{gstream}\ A \to A \times \langle \ell \mid \mathsf{gstream}\, A \rangle \,@\, t$$
$$\mathsf{gunfold}\, A = \mathsf{transport}\,(\mathsf{unfold}\,(\mathsf{gstream\_fun}\ A))$$

We are able to deduce the following equalities by using the fact that transport p is inverse to transport p^{-1}:

$$\mathsf{fold\_unfold} : (s : \mathsf{gstream}\, A) \to \mathsf{gfold}\, A\, (\mathsf{gunfold}\, A\, s) \equiv s \,@\, t$$
$$\mathsf{unfold\_fold} : (s : A \times \langle \ell \mid \mathsf{gstream}\, A \rangle) \to \mathsf{gunfold}\, A\, (\mathsf{gfold}\, A\, s) \equiv s \,@\, t$$

Using this we can define the familiar operations on guarded streams and prove their expected equations.

$$\mathsf{ghead} : \mathsf{gstream}\, A \to A$$
$$\_ : \mathsf{gtail}(\mathsf{gcons}\, a\, s) \equiv s$$
$$\_ : \mathsf{ghead}(\mathsf{gcons}\, a\, s) \equiv a$$

$$\mathsf{gtail} : \mathsf{gstream}\, A \to \langle \ell \mid \mathsf{gstream}\, A \rangle$$
$$\mathsf{gcons} : A \to \langle \ell \mid \mathsf{gstream}\, A \rangle \to \mathsf{gstream}\, A$$
$$\_ : \mathsf{gcons}\, (\mathsf{ghead} s)\, (\mathsf{gtail} s) \equiv s$$

With Löb induction, these definitions and equalities allow us to construct and work with *guarded* streams, which differ from coinductive streams in several important ways. For instance, the tail operation on guarded streams produces a guarded stream under a later which prevents us from writing an operation dropping every element of a guarded stream.

By making use of the other modalities of Fig. 4.4, we are able to define the type of *coinductive* streams. To do so, we will use the following operations:

$$\mathsf{comp}_{\gamma,\delta} : \langle \gamma \mid \langle \delta \mid A \rangle \rangle \to A \qquad \mathsf{comp}_{\gamma,\ell} : \langle \gamma \mid \langle \ell \mid A \rangle \rangle \to \langle \gamma \mid A \rangle$$

Both of these are instances of the general composition principle for modalities available in MTT. We now define streams as follows:

$$\mathsf{stream} : \mathsf{U} \to \mathsf{U} \,@\, s$$
$$\mathsf{stream}\, A = \langle \gamma \mid \mathsf{gstream}\, \langle \delta \mid A \rangle \rangle$$

$$\mathsf{head} : \mathsf{stream}\, A \to A$$
$$\mathsf{head}\, s =$$
$$\quad \mathtt{let}_{\mathsf{id}}\ \mathtt{mod}_{\gamma}(g) = s\ \mathtt{in}$$
$$\quad \mathsf{comp}_{\gamma,\delta}(\mathtt{mod}_{\gamma}(\mathsf{ghead}\, g))$$

$$\mathsf{tail} : \mathsf{stream}\, A \to \mathsf{stream}\, A$$
$$\mathsf{tail}\, s =$$
$$\quad \mathtt{let}_{\mathsf{id}}\ \mathtt{mod}_{\gamma}(g) = s\ \mathtt{in}$$
$$\quad \mathsf{comp}_{\gamma,\ell}(\mathtt{mod}_{\gamma}(\mathsf{gtail}\, g))$$

We emphasize that the type of coinductive streams lives at mode s, the mode modeled by sets. Intuitively, by taking the global sections of a guarded stream we obtain the normal coinductive stream [36]. Indeed, using guarded recursion in mode t, we are able to equip this type with a coiteration principle:

$$\mathsf{go} : (\delta \mid A : \mathsf{U})(\delta \mid S : \mathsf{U})(\delta \mid S \to A \times S) \to (\delta \mid S) \to \mathsf{gstream}\, \langle \delta \mid A \rangle \,@\, t$$
$$\mathsf{go}\, A\, S\, f = \mathsf{loeb}\, \lambda g\, s.\, \mathsf{gcons}(\mathsf{mod}_{\delta}(\pi_1\, (f\, s)), \mathsf{mod}_{\ell}(g\, (\pi_2\, (f\, s))))$$

$$\mathsf{coiter} : (A : \mathsf{U})(S : \mathsf{U}) \to (S \to A \times S) \to S \to \mathsf{stream}\, A \,@\, s$$
$$\mathsf{coiter}\, A\, S\, f\, s = \mathtt{mod}_{\gamma}(\mathsf{go}\, A\, S\, f\, s)$$

Informally, this coiteration scheme induces a map from any $(A \times -)$-coalgebra to $\mathsf{stream}\,A$.

It is natural to wonder whether $\mathsf{stream}\,A$ is the *final coalgebra* for $(A \times -)$. In the presence of equality reflection, this was established by Gratzer et al. [58]. To replay this proof in `mitten`, we would require two ingredients not presently available: function extensionality and modal extensionality. The first is unsurprising, so we focus on the second. Modalities do not necessarily preserve identity types and therefore in general we cannot have a function:

$$(\ell \mid \mathsf{Id}_A(a,b)) \to \mathsf{Id}_{\langle \ell | A \rangle}(\mathtt{mod}_\ell(a), \mathtt{mod}_\ell(b))$$

Such a map is crucial to establish arguments of equality by Löb induction like the finality of $\mathsf{stream}\,A$. Having said this, we emphasize that without disrupting normalization we can extend MTT with a crisp induction principle which enables us to construct such a map and prove it to be an equivalence [54]. In the presence of this additional structure—or a postulate to the same effect—we conjecture that $\mathsf{stream}\,A$ is the final coalgebra.

We conclude with a simple example of the coiteration: the stream of all natural numbers.

$$\begin{aligned}&\mathsf{nats} : \mathsf{stream}\,\mathtt{Nat}\\ &\mathsf{nats} = \mathsf{coiter}\,(\lambda n.\ (n, \mathtt{succ}(n)))\,0\end{aligned}$$

4.7 Related Work

Modal proof assistants have seen a great deal of attention in the last several years. We compare our work on `mitten` to several of the most closely related lines of research.

Proof assistants for a single modality There have been multiple attempts to extend proof assistants with a single specific modality. Notably Vezzosi [109] extends `Agda` with an idempotent comonad and Gratzer et al. [57] created a proof assistant based around a similar modality. Both of these proof assistants are closely related to `mitten`—indeed, the former may be encoded within `mitten`. Importantly, however, unlike these implementations `mitten` is not tied to a particular modal situation and can be easily adapted to accommodate other modalities.

Normalization for MTT In [54], Gratzer proves that MTT enjoys a normalization algorithm. While his proof avoids a number of technicalities by adopting a synthetic approach to normalization, this obstructs extracting an actual algorithm for use in implementation. We have taken this next step and, inspired by the synthetic proof of normalization, obtained an actual algorithm suitable for implementation in the particular case of preordered mode theories. Furthermore, while Gratzer works relative to the assumption that the ambient mode theory is decidable, we have isolated the precise requirements necessary on the mode theory and shown that they are sufficiently flexible to accommodate common mode theories.

Guarded recursion in Agda In Section 4.6 we discussed an instantiation of `mitten` for guarded recursion. For this specific case, an experimental Agda extension is available [104]. This extension implements a version of clocked cubical type theory [72]. This variant of guarded type theory offers finer-grained guarded programming by exposing multiple independent later modalities; these can be used to interleave guarded types without issue. Furthermore, clocked cubical type theory capitalizes on certain primitives of cubical type theory to expose some definitional equalities around Löb induction. Guarded cubical Agda builds upon Agda's existing facilities for interactive proof developments and the system has been used for non-trivial developments [83, 108].

As a consequence of this more intricate theory, however, the metatheory of guarded cubical Agda is far less developed than the theory of `mitten`. Moreover, the infrastructure of guarded cubical Agda is (necessarily) specialized to just one modal situation. While `mitten` is a more primitive system than guarded cubical Agda, it is therefore far more flexible and offers a theoretical framework for many modal systems rather than being specialized to one.

Sikkel Recently, Ceulemans et al. [33] have explored an alternative strategy for implementing MTT in Sikkel. Rather than constructing a custom proof assistant like `mitten`, they have provided a DSL for a simply-typed version of MTT within Agda. Within this DSL, one may construct terms in MTT which then compile to elements of an appropriate denotational semantics expressed within Agda. A major advantage of such an approach is the low startup cost: the full resources of the Agda proof assistant are available when working within Sikkel. By embedding within Agda, however, Sikkel's interface is less convenient and it is currently restricted only to simple types. Accordingly, we believe that a proof assistant designed for MTT from its inception offers a more promising route for serious modal programming.

Menkar Menkar [90] is an earlier attempt at a proof assistant for multimodal programming developed by Nuyts. It predates—and in fact partially inspires—MTT, but contains both theoretical and practical deficiencies which led to its development being suspended in 2019. Inspired by the advances in proof theory for multimodal type theory obtained since Menkar's development, both `mitten` and Sikkel are early attempts to develop a theoretically sound replacement for Menkar. While not as fully-featured as Menkar, `mitten` in particular is an attempt to develop a principled modal proof assistant.

4.8 Conclusions and future work

We contribute `mitten`, a flexible proof assistant which can be specialized to a wide range of modal type theories. We have designed normalization and type-checking algorithms for `mitten` based on recent advances in the metatheory of MTT [54]. Finally, we have argued for `mitten`'s utility by instantiating it to a mode theory

suitable for guarded recursion and constructing various classical examples of guarded programs.

Thus far, `mitten` is restricted to working with preordered mode theories. While this constitutes a large and important class of examples, it would be desirable to implement full MTT and allow for arbitrary 2-categories as mode theories. Such an extension, however, would require a more refined normalization algorithm.

In particular, in our algorithm we have taken advantage of the absence of distinct 2-cells to avoid annotating variables with modal coercions. This, in turn, preserves a crucial invariant of NbE: it is never necessary to explicitly substitute within a value. Indeed, in our style of NbE such substitutions are not even possible; our representation of closures essentially precludes them. We hope to generalize our approach to cover full MTT by incorporating some techniques recently used by Hu and Pientka [63] in a normalization algorithm for a particular modal type theory. Essentially, they enable a small amount of substitution to occur during the normalization algorithm; by carefully structuring the necessary modal substitutions they are able to adapt the standard normalization-by-evaluation to their setting. We hope to do the same in `mitten` by generalizing their approach to support multiple interacting modalities.

Appendix

Omitted proofs of chapter 3

This appendix contains proofs omitted from the main text.

Section 3.2

Proof of Lemma 3.2.1. The canonical map $A \to \forall\kappa.A$ maps a to $\lambda\kappa.a$. The map defined by application to the clock constant κ_0 defines a left inverse to this map. It suffices to show that this is also a right inverse, i.e., that $a[\kappa] = a[\kappa_0]$ for all $a : \forall\kappa.A$ and all κ. Since the canonical map $B \to \forall\kappa.B$ is assumed to be an equivalence, and application to κ_0 is a left inverse, it must also be a right inverse, so the corresponding property $b[\kappa] = b[\kappa_0]$ for all $b : \forall\kappa.B$. Applying this to $b = \lambda\kappa.i(a[\kappa])$ we get $i(a[\kappa]) = i(a[\kappa_0])$, which by assumption implies $a[\kappa] = a[\kappa_0]$. □

Section 3.3

Proof of Lemma 3.3.6. Using the classical definition of $\mathscr{D}$ in terms of finite maps, the equivalence $\mathscr{D}(A+1) \to 1 + [0,1) \times \mathscr{D}A$ would map f to $\star$ if $f(\star) = 1$ and otherwise to the pair $(f(\star), \frac{1}{1-f(\star)} \cdot g)$ where g is the restriction of f to A. In the special case of $A = \mathsf{Fin}(n)$, by Lemma 3.3.4, we can indeed define the equivalence like that and prove it an equivalence, also in CCTT. If we do that and transport the convex algebra structure from $\mathscr{D}(\mathsf{Fin}(n+1))$ to $1 + [0,1) \times \mathscr{D}(\mathsf{Fin}(n))$ we get a structure satisfying the following equalities.

$$\begin{aligned}
\star \oplus_p \star &= \star \\
(q,\mu) \oplus_p \star &= (pq + (1-p), \mu) \\
\star \oplus_p (r,\nu) &= (p + (1-p)r, \nu) \\
(q,\mu) \oplus_p (r,\nu) &= (pq + (1-p)r, \mu \oplus_{\left(\frac{p(1-q)}{p(1-q)+(1-p)(1-r)}\right)} \nu)
\end{aligned}$$

To prove the general statement, we will define a convex algebra on $1 + [0,1) \times \mathscr{D}A$ using the above equations, and prove that it defines the free convex algebra on $A+1$.

First, to see that the convex algebra operation satisfies the axioms of convex algebras, we use the fact that it does so in the case of $A = \mathsf{Fin}(n)$. To prove transitivity, for example, there are 8 cases to consider. One of them is

$$((r,\mu) \oplus_p (s,\nu)) \oplus_q (t,\rho) = (r,\mu) \oplus_{pq} \left((s,\nu) \oplus_{\frac{q-pq}{1-pq}} (t,\rho)\right) \tag{1}$$

for given $p,q,r,s,t : [0,1)$ and $\mu,\nu,\rho : \mathscr{D}(A)$. By Lemma 3.3.5 there exists an n, and $\mu',\nu',\rho' : \mathscr{D}(\mathsf{Fin}(n))$ as well as $f : \mathsf{Fin}(n) \to A$ such that $\mu = \mathscr{D}(f)(\mu')$, $\nu = \mathscr{D}(f)(\nu')$ and $\rho = \mathscr{D}(f)(\rho')$. (The lemma gives three different n, and different functions f but we can find a common domain as $n+n+n$ as in the proof of Lemma 3.3.5). Now, since (1) holds for μ',ν',ρ' we can apply $[0,1) \times \mathscr{D}(f)$ to both sides and get (1).

To see that this defines the free convex algebra structure on $A+1$, suppose B is a convex algebra and $f : A+1 \to B$. Let $g : \mathscr{D}A \to B$ be the unique extension of the

restriction of f to A, and define the extension $\overline{f}$ of f by the clauses

$$\overline{f}(\star) \triangleq f(\star) \qquad\qquad \overline{f}(p,\mu) \triangleq f(\star) \oplus_p g(\mu)$$

if $p > 0$ and $g(\mu)$ if $p = 0$. To show that this is a homomorphism, we must consider four cases. One is

$$\overline{f}((q,\mu) \oplus_p (r,\nu)) = \overline{f}(q,\mu) \oplus_p \overline{f}(r,\nu))$$

where (assuming q, r not zero) the left hand side unfolds to

$$\begin{aligned} &f(\star) \oplus_{pq+(1-p)r} g(\mu \oplus_{\left(\frac{p(1-q)}{p(1-q)+(1-p)(1-r)}\right)} \nu) \\ &= f(\star) \oplus_{pq+(1-p)r} \left(g(\mu) \oplus_{\left(\frac{p(1-q)}{p(1-q)+(1-p)(1-r)}\right)} g(\nu) \right) \end{aligned}$$

and the right hand side to

$$(f(\star) \oplus_q g(\mu)) \oplus_p (f(\star) \oplus_r g(\nu))$$

and now the case is easily verified using the technique of 23.

Finally, to show uniqueness, suppose $h : 1 + [0,1) \times \mathscr{D}A \to B$ is another homomorphism extending f. Then clearly $h(\star) = f(\star) = \overline{f}(\star)$, and since $\lambda\mu.(0,\mu)$ is a homomorphism from $\mathscr{D}A$ to $1 + [0,1) \times \mathscr{D}A$, we also get

$$h(0,\mu) = g(\mu) = \overline{f}(0,\mu)$$

by induction on μ. So finally since $(p,\mu) = \star \oplus_p (0,\mu)$ also $h(p,\mu) = \overline{f}(p,\mu)$ for any (p,μ). □

Section 3.4

Proof of Lemma 3.4.6. 1. By induction on n. For $n = 0$, note that $\mathsf{run}^n \nu = \nu$. So the statement simplifies to $\nu \rightsquigarrow \nu$, which holds by definition. For $n = n' + 1$, we consider the different cases for ν.

- If $\nu = \delta^{\forall} a$, then $\mathsf{run}^n \nu = \nu$ and the statement holds by definition.
- If $\nu = \mathsf{step}^{\forall} \nu'$, then we know $\nu \rightsquigarrow \nu'$ by definition of $\rightsquigarrow$. In addition, $\mathsf{run}^n \nu = \mathsf{run}^{n'} \nu'$, and we know from our IH that $\nu' \rightsquigarrow \mathsf{run}^{n'} \nu'$. Hence by transitivity of $\rightsquigarrow$: $\nu \rightsquigarrow \mathsf{run}^n \nu$.
- If $\nu = \nu_1 \oplus_p^{\forall} \nu_2$, then $\mathsf{run}^n \nu = \mathsf{run}^n \nu_1 \oplus_p^{\forall} \mathsf{run}^n \nu_2$, then we may assume that $\nu_1 \rightsquigarrow \mathsf{run}^n \nu_1$ and $\nu_2 \rightsquigarrow \mathsf{run}^n \nu_2$. Then by the choice axiom of $\rightsquigarrow$ also: $\nu \rightsquigarrow \mathsf{run}^n \nu_1 \oplus_p^{\forall} \mathsf{run}^n \nu_2 = \mathsf{run}^n \nu_1 \oplus_p^{\forall} \nu_2 = \mathsf{run}^n \nu$.

2. By induction on n. The base case $n = 0$ is trivial because $\mathsf{run}^0 \nu = \nu$ for all ν. For $n = n' + 1$, we go by induction on $\rightsquigarrow$.

- For the base case $v = v'$, the statement is trivial.
- For $v = \mathsf{step}^{\forall} v'$, we have $\mathsf{run}^n v = \mathsf{run}^{n'} v'$. By the first part of this Lemma, then $\mathsf{run}^n v \rightsquigarrow \mathsf{run}^n v'$.
- If there is a v'' such that $v \rightsquigarrow v''$ and $v'' \rightsquigarrow v'$, then by induction we may assume that $\mathsf{run}^n v \rightsquigarrow \mathsf{run}^n v''$ and $\mathsf{run}^n v'' \rightsquigarrow \mathsf{run}^n v'$. Then by transitivity of $\rightsquigarrow$ also $\mathsf{run}^n v \rightsquigarrow \mathsf{run}^n v'$.
- Lastly, if $v = v_1 \oplus_p^{\forall} v_2$ and $v' = v_1' \oplus_p^{\forall} v_2'$ such that $v_1 \rightsquigarrow v_1'$ and $v_2 \rightsquigarrow v_2'$, then by induction we may assume that also $\mathsf{run}^n v_1 \rightsquigarrow \mathsf{run}^n v_1'$, and similarly for v_2 and v_2'. Then also:

$$\begin{aligned}
\mathsf{run}^n v &= \mathsf{run}^n v_1 \oplus_p^{\forall} v_2 \\
&= \mathsf{run}^n v_1 \oplus_p^{\forall} \mathsf{run}^n v_2 \\
&\rightsquigarrow \mathsf{run}^n v_1' \oplus_p^{\forall} \mathsf{run}^n v_2' \\
&= \mathsf{run}^n v_1' \oplus_p^{\forall} \mathsf{run}^n v_2' \\
&= \mathsf{run}^n v'
\end{aligned}$$

3. By induction on $\rightsquigarrow$.
   - The base case for $v' = v$ is trivial.
   - For $v = \mathsf{step}^{\forall} v'$, we have that $\mathsf{run}^1 v = v'$, and hence $v' \rightsquigarrow \mathsf{run}^1 v$.
   - If there is a v'' such that $v \rightsquigarrow v''$ and $v'' \rightsquigarrow v'$, then by induction there is an m_1 such that $v'' \rightsquigarrow \mathsf{run}^{m_1} v$ and there is an m_2 such that $v' \rightsquigarrow \mathsf{run}^{m_2} v''$. Then the second part of this lemma, also $\mathsf{run}^{m_2} v'' \rightsquigarrow \mathsf{run}^{m_2}\mathsf{run}^{m_1} v = \mathsf{run}^{m_1+m_2} v$, and hence by transitivity: $v' \rightsquigarrow \mathsf{run}^{m_1+m_2} v$.
   - Lastly, if $v = v_1 \oplus_p^{\forall} v_2$ and $v' = v_1' \oplus_p^{\forall} v_2'$ such that $v_1 \rightsquigarrow v_1'$ and $v_2 \rightsquigarrow v_2'$, then by induction there is an m_1 such that $v_1' \rightsquigarrow \mathsf{run}^{m_1} v_1$ and an m_2 such that $v_2' \rightsquigarrow \mathsf{run}^{m_2} v_2$. Let $n = max(m_1, m_2)$. Then by the first part of this lemma: $v_1' \rightsquigarrow \mathsf{run}^n v_1$ and $v_2' \rightsquigarrow \mathsf{run}^n v_2$. So then also $v' \rightsquigarrow \mathsf{run}^n v$.

□

Proof of Lemma 3.4.7. By induction on $\rightsquigarrow$:

- The first base case is obvious by reflexivity.
- For the second base case, note that:

$$\overline{f}(\mathsf{step}^{\forall} v) = \mathsf{step}^{\forall}(\overline{f}(v)) \rightsquigarrow \overline{f}(v)$$

- If $v \rightsquigarrow v'$ came from the transitivity axiom, then there is a v'' such that $v \rightsquigarrow v''$ and $v'' \rightsquigarrow v'$. By the induction hypothesis we know that then also $\overline{f}(v) \rightsquigarrow \overline{f}(v'')$ and $\overline{f}(v'') \rightsquigarrow \overline{f}(v')$. Then by transitivity of $\rightsquigarrow$: $\overline{f}(v) \rightsquigarrow \overline{f}(v')$.

- If $v = v_1 \oplus_p^\forall v_2$ and $v' = v_1' \oplus_p^\forall v_2'$, where $v_1 \rightsquigarrow v_1'$ and $v_2 \rightsquigarrow v_2'$, then by the induction hypothesis: $\overline{f}(v_1) \rightsquigarrow \overline{f}(v_1')$ and $\overline{f}(v_2) \rightsquigarrow \overline{f}(v_2')$. Then:

$$\begin{aligned}\overline{f}(v) &= \overline{f}(v_1 \oplus_p^\forall v_2)\\ &= (\overline{f}(v_1)) \oplus_p^\forall (\overline{f}(v_2))\\ &\rightsquigarrow (\overline{f}(v_1')) \oplus_p^\forall (\overline{f}(v_2'))\\ &= \overline{f}(v_1' \oplus_p^\forall v_2')\\ &= \overline{f}(v')\end{aligned}$$

□

Proof of Lemma 3.4.8. 1. For the fist statement, we prove that $\mathsf{PT}_n(v) \leq \mathsf{PT}_{n+1}(v)$ by induction on n. For $n = 0$, we do a case analysis of v:

- If $v = \delta^\forall a$, then by definition $\mathsf{PT}_0(\delta^\forall a) = 1$, and $\mathsf{PT}_1(\delta^\forall a) = \mathsf{PT}_0(\mathsf{run}(\delta^\forall a)) = \mathsf{PT}_0(\delta^\forall a) = 1$. So in this case we have equality.
- If $v = \mathsf{step}^\forall v'$, then by definition $\mathsf{PT}_0(\mathsf{step}^\forall v') = 0$ and $\mathsf{PT}_1(\mathsf{step}^\forall v') = \mathsf{PT}_0(\mathsf{run}(\mathsf{step}^\forall v')) = \mathsf{PT}_0(v') \geq 0$.
- If $v = v_1 \oplus_p^\forall v_2$, then we may assume that $\mathsf{PT}_0(v_1) \leq \mathsf{PT}_1(v_1)$ and $\mathsf{PT}_0(v_2) \leq \mathsf{PT}_1(v_2)$. Then also:

$$\begin{aligned}\mathsf{PT}_0(v_1 \oplus_p^\forall v_2) &= \mathsf{PT}_0(v_1) \oplus_p^\forall \mathsf{PT}_0(v_2)\\ &\leq \mathsf{PT}_1(v_1) \oplus_p^\forall \mathsf{PT}_1(v_2)\\ &= \mathsf{PT}_0(\mathsf{run}(v_1)) \oplus_p^\forall \mathsf{PT}_0(\mathsf{run}(v_2))\\ &= \mathsf{PT}_0(\mathsf{run}(v_1 \oplus_p^\forall v_2))\\ &= \mathsf{PT}_1(v_1 \oplus_p^\forall v_2)\end{aligned}$$

For $n = n' + 1$, notice that $\mathsf{PT}_{n+1}(v) = \mathsf{PT}_n(\mathsf{run}(v))$. We again go by case analysis of v:

- If $v = \delta^\forall a$ for some $a : A$, then $\mathsf{PT}_n(\delta^\forall a) = \mathsf{PT}_{n+1}(\delta^\forall a) = 1$.
- If $v = \mathsf{step}^\forall v'$, then $\mathsf{PT}_n(\mathsf{step}^\forall v') = \mathsf{PT}_{n'}(\mathsf{run}(\mathsf{step}^\forall v')) = \mathsf{PT}_{n'}(v')$, and $\mathsf{PT}_{n+1}(\mathsf{step}^\forall v') = \mathsf{PT}_n(\mathsf{run}(\mathsf{step}^\forall v')) = \mathsf{PT}_n(v')$. By the induction hypothesis for n, we know that $\mathsf{PT}_{n'}(v') \leq \mathsf{PT}_n(v')$, and hence $\mathsf{PT}_n(\mathsf{step}^\forall v') \leq \mathsf{PT}_{n+1}(\mathsf{step}^\forall v')$.
- If $v = v_1 \oplus_p^\forall v_2$, then $\mathsf{PT}_n()$ distributes through the sum and via a similar reasoning as above we get to the conclusion.

2. The second statement of this Lemma we also prove by induction on n. For $n = 0$ we go by induction on $\rightsquigarrow$.

- If $v = v'$, then the statement is trivially true.

- If $\nu = \mathsf{step}^{\forall}\nu'$, by definition $\mathsf{PT}_0(\nu) = 0$, and hence: $\mathsf{PT}_0(\nu) \leq \mathsf{PT}_0(\nu')$.
- If there is a ν'' such that $\nu \rightsquigarrow \nu''$ and $\nu'' \rightsquigarrow \nu'$, then by the induction hypothesis we have $\mathsf{PT}_0(\nu) \leq \mathsf{PT}_0(\nu'')$ and $\mathsf{PT}_0(\nu'') \leq \mathsf{PT}_0(\nu')$. Then by transitivity of $\leq$, we also have $\mathsf{PT}_0(\nu) \leq \mathsf{PT}_0(\nu')$.
- Lastly, if $\nu = \nu_1 \oplus_p^{\forall} \nu_2$ and $\nu' = \nu_1' \oplus_p^{\forall} \nu_2'$ such that $\nu_1 \rightsquigarrow \nu_1'$ and $\nu_2 \rightsquigarrow \nu_2'$, then by the induction hypothesis we have $\mathsf{PT}_0(\nu_1) \leq \mathsf{PT}_0(\nu_1')$ and $\mathsf{PT}_0(\nu_2) \leq \mathsf{PT}_0(\nu_2')$. Then:

$$\begin{aligned}\mathsf{PT}_0(\nu) &= p \cdot \mathsf{PT}_0(\nu_1) + (1-p) \cdot \mathsf{PT}_0(\nu_2) \\ &\leq p \cdot \mathsf{PT}_0(\nu_1') + (1-p) \cdot \mathsf{PT}_0(\nu_2') \\ &= \mathsf{PT}_0(\nu').\end{aligned}$$

For $n = n' + 1$, we again go by induction on $\rightsquigarrow$. The cases of $\nu = \nu'$, transitivity, and sum are the same as above (note that $\mathsf{PT}_n()$ distributes over $\_ \oplus_p^{\forall} \_$ because both run and $\mathsf{PT}_0()$ do so). The last case, $\nu = \mathsf{step}^{\forall}\nu'$, we prove here. By definition, $\mathsf{PT}_n(\nu) = \mathsf{PT}_{n'}(\nu')$. Then by the first part of this lemma, we have $\mathsf{PT}_{n'}(\nu') \leq \mathsf{PT}_n(\nu')$, and hence: $\mathsf{PT}_n(\nu) \leq \mathsf{PT}_n(\nu')$.

□

Section 3.5

Proof of Example 29. Let

$$E_V \triangleq \lambda(\mathsf{lam}\ x.M).\mathsf{step}^{\kappa}(\lambda(\alpha : \kappa).\mathsf{eval}^{\kappa}(M[V/x]))$$

Note that

$$\mathsf{eval}^{\kappa}(\mathsf{Y}f) = \Delta^{\kappa}(\mathsf{eval}^{\kappa}(\mathsf{lam}\ x.(e_f(\mathsf{fold}\, e_f))x))$$

and

$$\mathsf{eval}^{\kappa}(e_f(\mathsf{fold}\, e_f)) = (\Delta^{\kappa})^3(\mathsf{eval}^{\kappa}(f(\mathsf{lam}\ x.(e_f(\mathsf{fold}\, e_f)x))))$$

Therefore,

$$\begin{aligned}&\mathsf{eval}^{\kappa}((\mathsf{Y}f)(V)) \\ &= (\Delta^{\kappa})^2(\mathsf{eval}^{\kappa}(e_f(\mathsf{fold}\, e_f))V)) \\ &= (\Delta^{\kappa})^5(\mathsf{eval}^{\kappa}(f(\mathsf{lam}\ x.(e_f(\mathsf{fold}\, e_f)x)) >>=^{\kappa} E_V) \\ &= (\Delta^{\kappa})^5(\delta^{\kappa}(f) >>=^{\kappa} \lambda(\mathsf{lam}\ Z.P).\mathsf{eval}^{\kappa}(\mathsf{lam}\ x.(e_f(\mathsf{fold}\, e_f)x)) \\ &\qquad >>=^{\kappa} \lambda U.\Delta^{\kappa}(\mathsf{eval}^{\kappa}(P[U/z]) >>=^{\kappa} E_V)) \\ &= (\Delta^{\kappa})^4(\delta^{\kappa}(f) >>=^{\kappa} \lambda(\mathsf{lam}\ Z.P).\Delta^{\kappa}.\mathsf{eval}^{\kappa}(\mathsf{lam}\ x.(e_f(\mathsf{fold}\, e_f)x)) \\ &\qquad >>=^{\kappa} \lambda U.\Delta^{\kappa}(\mathsf{eval}^{\kappa}(P[U/z]) >>=^{\kappa} E_V)) \\ &= (\Delta^{\kappa})^4(\delta^{\kappa}(f) >>=^{\kappa} \lambda(\mathsf{lam}\ Z.P).\mathsf{eval}^{\kappa}(\mathsf{Y}f) \\ &\quad >>=^{\kappa} \lambda U.\Delta^{\kappa}(\mathsf{eval}^{\kappa}(P[U/z]) >>=^{\kappa} E_V)) \\ &= (\Delta^{\kappa})^4(\mathsf{eval}^{\kappa}((f(\mathsf{Y}f))) >>=^{\kappa} E_V)\end{aligned}$$

$$= (\Delta^\kappa)^4(\mathsf{eval}^\kappa((f(\mathsf{Y}f))(V)))$$

This completes the proof.

□

Section 3.6

Lemma .0.1 (Substitution Lemma)**.** *For any well-typed term* $\Gamma.(x:\sigma) \vdash M : \tau$ *as well as every well typed value* $\Gamma \vdash V : \sigma$ *we have*

$$[\![M[V/x]]\!]^\kappa_\rho = [\![M]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}$$

Proof. We proceed by induction on term derivations.

Case: $M = x$

Then

$$[\![x[V/x]]\!]^\kappa_\rho = [\![V]\!]^\kappa_\rho = [\![x]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}$$

Case: $\Gamma \vdash \mathsf{lam}\ x.M' : \tau_1 \to \tau_2$

Assume without loss of generality that $y \neq x$. Otherwise, use an α-equivalent term where this condition is true. Consequently, we have that

$$\begin{aligned}
&[\![\mathsf{lam}\ x.M'[V/y]]\!]^\kappa_\rho \\
&= \delta^\kappa([\![\mathsf{lam}\ x.(M'[V/y])]\!]^{\mathsf{Val},\kappa}_\rho) \\
&= \delta^\kappa(\lambda(v:[\![\tau_1]\!]^\kappa).[\![M'[V/y]]\!]^\kappa_{\rho,x\mapsto v}) \\
&= \delta^\kappa\left(\lambda(v:[\![\tau_1]\!]^\kappa).[\![M']\!]^\kappa_{\rho,x\mapsto v,y\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}\right) \\
&= \delta^\kappa\left([\![\mathsf{lam}\ x.M']\!]^{\mathsf{Val},\kappa}_{\rho,y\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}\right) \\
&= [\![\mathsf{lam}\ x.M']\!]^\kappa_{\rho,y\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}
\end{aligned}$$

Case: $M = MN$

We have that

$$\begin{aligned}
&[\![MN[V/x]]\!]^\kappa_\rho \\
&= [\![M[V/x]]\!]^\kappa_\rho \cdot [\![N[V/x]]\!]^\kappa_\rho \\
&= [\![M]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho} \cdot [\![N]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho} \\
&= [\![MN]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}
\end{aligned}$$

Case: $M = \mathsf{fold}\,M'$

We have that

$$[\![\mathsf{fold}\,M'[V/x]]\!]^\kappa_\rho$$

$$
\begin{aligned}
&= \mathscr{D}(\mathsf{next}^\kappa)\left([\![M'[V/x]]\!]^\kappa_\rho\right)\\
&= \mathscr{D}(\mathsf{next}^\kappa)\left([\![M']\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}\right)\\
&= [\![\mathsf{fold}\,M']\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}
\end{aligned}
$$

Case: $M = \mathsf{unfold}\,M'$
We have

$$
\begin{aligned}
&[\![\mathsf{unfold}\,M'[V/x]]\!]^\kappa_\rho\\
&= [\![M'[V/x]]\!]^\kappa_\rho >>=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda\alpha.\delta^\kappa(v[\alpha]))\\
&= [\![M']\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho} >>=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda\alpha.\delta^\kappa(v[\alpha]))\\
&= [\![\mathsf{unfold}\,M']\!]^\kappa_{\rho,x\mapsto[\![V]\!]^{\mathsf{Val},\kappa}_\rho}
\end{aligned}
$$

Case: $M = \mathsf{choice}^q(N_1,N_2)$ We have

$$
\begin{aligned}
&[\![\mathsf{choice}^q(N_1,N_2)[V/x]]\!]^\kappa_\rho\\
&= [\![N_1[V/x]]\!]^\kappa_\rho \oplus^\kappa_q [\![N_2[V/x]]\!]^\kappa_\rho\\
&= [\![N_1]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^\kappa_\rho} \oplus^\kappa_q [\![N_2]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^\kappa_\rho}\\
&= [\![\mathsf{choice}^q(N_1,N_2)]\!]^\kappa_{\rho,x\mapsto[\![V]\!]^\kappa_\rho}
\end{aligned}
$$

We leave the remaining cases to the reader. □

Relating Syntactic and Semantic Values

It is easy to show that meta-level eliminators for values such as $\underline{\pi_1} : \mathsf{Val}_{\sigma\times\tau} \to \mathsf{Val}_\sigma$ and $\underline{\pi_2} : \mathsf{Val}_{\sigma\times\tau} \to \mathsf{Val}_\tau$ commute with $[\![-]\!]^{\mathsf{Val},\kappa}$.

Lemma .0.2. *For all values* $\cdot \vdash V : \sigma\times\tau$ *and* $\cdot \vdash \underline{n} : \mathsf{Nat}$ *we have*

$$
\begin{aligned}
[\![\underline{\pi_1}V]\!]^{\mathsf{Val},\kappa} &= \mathsf{pr}_1[\![V]\!]^{\mathsf{Val},\kappa} & [\![\underline{\pi_2}V]\!]^{\mathsf{Val},\kappa} &= \mathsf{pr}_2[\![V]\!]^{\mathsf{Val},\kappa}\\
[\![\mathbf{suc}\ \underline{n}]\!]^{\mathsf{Val},\kappa} &= \mathbf{suc}[\![\underline{n}]\!]^{\mathsf{Val},\kappa} & [\![\mathbf{pred}\ \underline{n}]\!]^{\mathsf{Val},\kappa} &= \mathbf{pred}[\![\underline{n}]\!]^{\mathsf{Val},\kappa}
\end{aligned}
$$

Proof. **Case**: $\underline{\pi_1}V$
We have that $[\![\underline{\pi_1}(\langle V,W\rangle)]\!]^{\mathsf{Val},\kappa} = [\![V]\!]^{\mathsf{Val},\kappa} = \mathsf{pr}_1([\![\langle V,W\rangle]\!]^{\mathsf{Val},\kappa})$
Case: **pred** $\underline{n}$

$$
\begin{aligned}
[\![\mathbf{pred}\ \underline{n}]\!]^{\mathsf{Val},\kappa} &= \underline{[\![\mathbf{max}(0,n-1)]\!]^{\mathsf{Val},\kappa}} = \mathbf{max}(0,[\![\underline{n}]\!]^{\mathsf{Val},\kappa}-1)\\
&= \mathbf{pred}[\![\underline{n}]\!]^{\mathsf{Val},\kappa}
\end{aligned}
$$

The remaining cases are similar. □

Lemma .0.3. *For any meta-level type A and terms $a, b : A$ we we have*

$$\begin{cases} \underline{0} & \mapsto a \\ \underline{n+1} & \mapsto b \end{cases} = \left(\begin{cases} 0 & \mapsto a \\ n+1 & \mapsto b \end{cases} \right) \circ [\![-]\!]^{\mathsf{Val},\kappa}$$

as functions of the type $\mathsf{Val}_{\mathsf{Nat}} \to A$

Proof. This is a direct consequence of the fact that $\mathsf{Val}_{\mathsf{Nat}}$ and $\mathscr{N}$ are isomorphic. □

Lemma .0.4. *For functions $f : [\![\sigma]\!]^\kappa \to A$ and $g : [\![\tau]\!]^\kappa \to A$ we have*

$$\begin{cases} \mathsf{inl}\, V & \mapsto f([\![V]\!]^{\mathsf{Val},\kappa}) \\ \mathsf{inr}\, V & \mapsto g([\![V]\!]^{\mathsf{Val},\kappa}) \end{cases} = \begin{cases} \mathbf{inl}\, v & \mapsto f \\ \mathbf{inr}\, v & \mapsto g \end{cases} \circ [\![-]\!]^{\mathsf{Val},\kappa} : \mathsf{Val}_{\sigma+\tau} \to A$$

Proof of Theorem 3.6.1. First, assume the guarded hypothesis

$$\forall M : \mathsf{Tm}_\sigma.(\triangleright(\alpha : \kappa).(\mathsf{eval}^\kappa M >>=^\kappa \delta^\kappa \circ [\![-]\!]^{\mathsf{Val},\kappa}) = [\![M]\!]^\kappa).$$

We proceed with case analysis of term derivations.

Case: $M = V$ The value cases are all the same:

$$\begin{aligned} \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa V) &= \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)(\delta^\kappa(V)) \\ &= \delta^\kappa([\![V]\!]^{\mathsf{Val},\kappa}) \\ &= [\![V]\!]^\kappa \end{aligned}$$

Case: $M = \mathsf{suc}\, M'$

$$\begin{aligned} \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa(\mathsf{suc}\, M')) &= \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{D}^\kappa(\mathbf{suc})\,(\mathsf{eval}^\kappa M')\right) \\ &= \mathsf{D}^\kappa\left([\![\mathbf{suc}(-)]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa M') \\ &= \mathsf{D}^\kappa\left(\mathbf{suc}([\![-]\!]^{\mathsf{Val},\kappa})\right)(\mathsf{eval}^\kappa M') \\ &= \mathsf{D}^\kappa(\mathbf{suc})\left(\mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa M')\right) \\ &= \mathsf{D}^\kappa(\mathbf{suc})\left([\![M']\!]^\kappa\right) \\ &= [\![\mathsf{suc}\, M']\!]^\kappa \end{aligned}$$

Case: $M = \mathsf{pred}\, M'$

$$\begin{aligned} \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa(\mathsf{pred}\, M')) &= \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{D}^\kappa(\mathbf{pred})\,(\mathsf{eval}^\kappa M')\right) \\ &= \mathsf{D}^\kappa\left([\![\mathbf{pred}(-)]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa M') \\ &= \mathsf{D}^\kappa\left(\mathbf{pred}([\![-]\!]^{\mathsf{Val},\kappa})\right)(\mathsf{eval}^\kappa M') \\ &= \mathsf{D}^\kappa(\mathbf{pred})\left(\mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa M')\right) \end{aligned}$$

$$\begin{aligned}&= \mathsf{D}^\kappa(\mathbf{pred})\left([\![M']\!]^\kappa\right)\\ &= [\![\mathsf{pred}\,M']\!]^\kappa\end{aligned}$$

Case: $M = \mathsf{ifz}(L,M',N)$

$$\begin{aligned}&\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa(\mathsf{ifz}(L,M',N))\right)\\ &= \mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa L >>=^\kappa \begin{cases}\underline{0}\mapsto \mathsf{eval}^\kappa(M')\\ \underline{n+1}\mapsto \mathsf{eval}^\kappa(N)\end{cases}\right)\\ &= \left(\mathsf{eval}^\kappa L >>=^\kappa \begin{cases}\underline{0}\mapsto \mathsf{D}^\kappa\big([\![-]\!]^{\mathsf{Val},\kappa}\big)\left(\mathsf{eval}^\kappa(M')\right)\\ \underline{n+1}\mapsto \mathsf{D}^\kappa\big([\![-]\!]^{\mathsf{Val},\kappa}\big)\left(\mathsf{eval}^\kappa(N)\right)\end{cases}\right)\\ &= \mathsf{eval}^\kappa L >>=^\kappa \begin{cases}\underline{0}\mapsto [\![M']\!]^\kappa\\ \underline{n+1}\mapsto [\![N]\!]^\kappa\end{cases}\\ &= \mathsf{eval}^\kappa L >>=^\kappa \lambda\underline{n}.\mathsf{match}\ [\![\underline{n}]\!]^{\mathsf{Val},\kappa}\ \mathsf{with}\begin{cases}0\mapsto [\![M']\!]^\kappa\\ n+1\mapsto [\![N]\!]^\kappa\end{cases}\\ &= (\mathsf{eval}^\kappa L >>=^\kappa \delta^\kappa\circ[\![-]\!]^{\mathsf{Val},\kappa}) >>=^\kappa \begin{cases}0\mapsto [\![M']\!]^\kappa\\ n+1\mapsto [\![N]\!]^\kappa\end{cases}\\ &= [\![\mathsf{ifz}(L,M',N)]\!]^\kappa\end{aligned}$$

Case: $M = \langle M',N\rangle$

$$\begin{aligned}&\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa\langle M',N\rangle\right)\\ &= \mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa M' >>=^\kappa \lambda V.(\mathsf{eval}^\kappa N >>=^\kappa \lambda W.\delta^\kappa(\langle V,W\rangle))\right)\\ &= \mathsf{eval}^\kappa M' >>=^\kappa \lambda V.\left(\mathsf{eval}^\kappa N >>=^\kappa \lambda W.\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\delta^\kappa(\langle V,W\rangle)\right)\right)\\ &= \mathsf{eval}^\kappa M' >>=^\kappa \lambda V.\mathsf{eval}^\kappa N >>=^\kappa \lambda W.\delta^\kappa([\![\langle V,W\rangle]\!]^{\mathsf{Val},\kappa})\\ &= \mathsf{eval}^\kappa M' >>=^\kappa \lambda V.\mathsf{eval}^\kappa N >>=^\kappa \lambda W.\delta^\kappa(([\![V]\!]^{\mathsf{Val},\kappa},[\![W]\!]^{\mathsf{Val},\kappa}))\\ &= \begin{aligned}&(\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa M'\right)) >>=^\kappa \lambda v.(\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa N\right))\\ &\qquad\qquad >>=^\kappa \lambda w.\delta^\kappa((v,w))\end{aligned}\\ &= [\![M']\!]^\kappa >>=^\kappa \lambda v.[\![N]\!]^\kappa >>=^\kappa \lambda w.\delta^\kappa((v,w))\\ &= [\![\langle M',N\rangle]\!]^\kappa\end{aligned}$$

Case: $M = \mathsf{fst}\,M'$

$$\begin{aligned}&\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa(\mathsf{fst}\,M')\right)\\ &= \mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{D}^\kappa(\underline{\pi_1})\left(\mathsf{eval}^\kappa M'\right)\right)\\ &= \mathsf{D}^\kappa\Big([\![\underline{\pi_1}(-)]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa M'\right)\end{aligned}$$

$$
\begin{aligned}
&= \mathsf{D}^\kappa(\mathsf{pr}_1)\left(\mathsf{D}^\kappa\Big(\llbracket - \rrbracket^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa M'\right)\right)\\
&= \mathsf{D}^\kappa(\mathsf{pr}_1)\left(\llbracket M' \rrbracket^\kappa\right)\\
&= \llbracket \mathsf{fst}\, M' \rrbracket^\kappa
\end{aligned}
$$

Case: $M = \mathsf{snd}\, M'$

$$
\begin{aligned}
&\mathsf{D}^\kappa\Big(\llbracket - \rrbracket^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa(\mathsf{snd}\, M')\right)\\
&= \mathsf{D}^\kappa\Big(\llbracket - \rrbracket^{\mathsf{Val},\kappa}\Big)\left(\mathsf{D}^\kappa(\underline{\pi_2})\left(\mathsf{eval}^\kappa M'\right)\right)\\
&= \mathsf{D}^\kappa\Big(\llbracket \underline{\pi_2}(-) \rrbracket^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa M'\right)\\
&= \mathsf{D}^\kappa(\mathsf{pr}_2)\left(\mathsf{D}^\kappa\Big(\llbracket - \rrbracket^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^\kappa M'\right)\right)\\
&= \mathsf{D}^\kappa(\mathsf{pr}_2)\left(\llbracket M' \rrbracket^\kappa\right)\\
&= \llbracket \mathsf{snd}\, M' \rrbracket^\kappa
\end{aligned}
$$

Case: $M = MN$

We start by unfolding the definition of $(\mathsf{eval}^\kappa MN) >\!\!>\!\!=^\kappa \delta^\kappa \circ \llbracket - \rrbracket^{\mathsf{Val},\kappa}$ and use associativity as well as the definition of $>\!\!>\!\!=^\kappa$ to get

$$
\begin{aligned}
&\big(\mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\; x.M').\mathsf{eval}^\kappa N >\!\!>\!\!=^\kappa\\
&\qquad \lambda V.(\Delta^\kappa(\mathsf{eval}^\kappa(M'[V/x])))\big) >\!\!>\!\!=^\kappa \delta^\kappa \circ \llbracket - \rrbracket^{\mathsf{Val},\kappa}\\
&= \begin{pmatrix}\mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\; x.M').\mathsf{eval}^\kappa N\\ \qquad >\!\!>\!\!=^\kappa \lambda V.(\Delta^\kappa\big(\mathsf{eval}^\kappa(M'[V/x])\big) >\!\!>\!\!=^\kappa \delta^\kappa \circ \llbracket - \rrbracket^{\mathsf{Val},\kappa})\end{pmatrix}\\
&= \begin{pmatrix}\mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\; x.M').\mathsf{eval}^\kappa N\\ \qquad >\!\!>\!\!=^\kappa \lambda V.\Delta^\kappa\left(\mathsf{eval}^\kappa(M'[V/x]) >\!\!>\!\!=^\kappa \delta^\kappa \circ \llbracket - \rrbracket^{\mathsf{Val},\kappa}\right)\end{pmatrix}\\
&= \begin{pmatrix}\mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\; x.M').\mathsf{eval}^\kappa N\\ \qquad >\!\!>\!\!=^\kappa \lambda V.(\mathsf{step}^\kappa(\lambda\alpha.\llbracket M'[V/x] \rrbracket^\kappa_\rho))\end{pmatrix}
\end{aligned}
$$

In the last step we applied the guarded hypothesis. Next, we use the substitution lemma for terms and recall the definition of semantic function type values, which leaves us with

$$
\begin{aligned}
&= \mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\; x.M').\mathsf{eval}^\kappa N >\!\!>\!\!=^\kappa \lambda W.(\Delta^\kappa(\llbracket\, M' \rrbracket^\kappa_{\llbracket W \rrbracket^{\mathsf{Val},\kappa}}))\\
&= \begin{pmatrix}\mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda(\mathsf{lam}\; x.M').\mathsf{eval}^\kappa N\\ \qquad >\!\!>\!\!=^\kappa \lambda W.(\Delta^\kappa(\llbracket \mathsf{lam}\; x.M' \rrbracket^{\mathsf{Val},\kappa}(\llbracket W \rrbracket^{\mathsf{Val},\kappa})))\end{pmatrix}\\
&= \mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \lambda V.\mathsf{eval}^\kappa N >\!\!>\!\!=^\kappa \lambda W.(\Delta^\kappa(\llbracket V \rrbracket^{\mathsf{Val},\kappa}(\llbracket W \rrbracket^{\mathsf{Val},\kappa})))
\end{aligned}
$$

In the last equation, we simply omitted the superfluous case analysis of function values. We now use the functoriality of D^κ to get

$$
= (\mathsf{eval}^\kappa M >\!\!>\!\!=^\kappa \llbracket - \rrbracket^{\mathsf{Val},\kappa}) >\!\!>\!\!=^\kappa \lambda v.(\mathsf{eval}^\kappa N >\!\!>\!\!=^\kappa \llbracket - \rrbracket^{\mathsf{Val},\kappa})
$$

$$>>=^{\kappa} \lambda w.\Delta^{\kappa}(vw)$$

which by the induction hypothesis gives

$$\begin{aligned}
&= [\![M]\!]^{\kappa} >>=^{\kappa} \lambda v.[\![N]\!]^{\kappa} >>=^{\kappa} \lambda w.\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).vw)\\
&= [\![M]\!]^{\kappa} \cdot [\![N]\!]^{\kappa}\\
&= [\![MN]\!]^{\kappa}
\end{aligned}$$

Case: $M = \mathsf{inl}\, M'$

$$\begin{aligned}
&\mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^{\kappa}(\mathsf{inl}\, M')\right)\\
&= \mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{D}^{\kappa}(\mathsf{inl})\left(\mathsf{eval}^{\kappa} M'\right)\right)\\
&= \mathsf{D}^{\kappa}\Big([\![\mathsf{inl}\,(-)]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^{\kappa} M'\right)\\
&= \mathsf{D}^{\kappa}(\mathbf{inl})\left(\mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^{\kappa} M'\right)\right)\\
&= \mathsf{D}^{\kappa}(\mathbf{inl})\left([\![M']\!]^{\kappa}\right)\\
&= [\![\mathsf{inl}\, M']\!]^{\kappa}
\end{aligned}$$

Case: $M = \mathsf{inr}\, M'$

$$\begin{aligned}
&\mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^{\kappa}(\mathsf{inr}\, M')\right)\\
&= \mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{D}^{\kappa}(\mathsf{inr})\left(\mathsf{eval}^{\kappa} M'\right)\right)\\
&= \mathsf{D}^{\kappa}\Big([\![\mathsf{inr}\,(-)]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^{\kappa} M'\right)\\
&= \mathsf{D}^{\kappa}(\mathbf{inr})\left(\mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^{\kappa} M'\right)\right)\\
&= \mathsf{D}^{\kappa}(\mathbf{inr})\left([\![M']\!]^{\kappa}\right)\\
&= [\![\mathsf{inr}\, M']\!]^{\kappa}
\end{aligned}$$

Case: $M = \mathsf{case}(L, x.M, y.N)$

We proceed similarly to the function application case and first unfold the definition of $\mathsf{D}^{\kappa}\big([\![-]\!]^{\mathsf{Val},\kappa}\big)\,(\mathsf{eval}^{\kappa}\mathsf{case}(L, x.M, y.N))$

$$\begin{aligned}
&\mathsf{D}^{\kappa}\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\left(\mathsf{eval}^{\kappa} L >>=^{\kappa} \begin{cases} \mathsf{inl}\, V \mapsto (\Delta^{\kappa}\mathsf{eval}^{\kappa}(M[V/x])) \\ \mathsf{inr}\, V \mapsto (\Delta^{\kappa}\mathsf{eval}^{\kappa}(N[V/x])) \end{cases}\right)\\
&= \mathsf{eval}^{\kappa} L >>=^{\kappa} \begin{cases} \mathsf{inl}\, V \mapsto \mathsf{D}^{\kappa}\big([\![-]\!]^{\mathsf{Val},\kappa}\big)\,(\Delta^{\kappa}(\mathsf{eval}^{\kappa}(M[V/x]))) \\ \mathsf{inr}\, V \mapsto \mathsf{D}^{\kappa}\big([\![-]\!]^{\mathsf{Val},\kappa}\big)\,(\Delta^{\kappa}(\mathsf{eval}^{\kappa}(N[V/x]))) \end{cases}\\
&= \mathsf{eval}^{\kappa} L >>=^{\kappa} \begin{cases} \mathsf{inl}\, V \mapsto \Delta^{\kappa}(\mathsf{D}^{\kappa}\big([\![-]\!]^{\mathsf{Val},\kappa}\big)\,(\mathsf{eval}^{\kappa}(M[V/x]))) \\ \mathsf{inr}\, V \mapsto \Delta^{\kappa}(\mathsf{D}^{\kappa}\big([\![-]\!]^{\mathsf{Val},\kappa}\big)\,(\mathsf{eval}^{\kappa}(N[V/x]))) \end{cases}\\
&= \mathsf{eval}^{\kappa} L >>=^{\kappa} \begin{cases} \mathsf{inl}\, V \mapsto (\Delta^{\kappa}[\![M[V/x]]\!]^{\kappa}) \\ \mathsf{inr}\, V \mapsto (\Delta^{\kappa}[\![N[V/x]]\!]^{\kappa}) \end{cases}
\end{aligned}$$

$$
\begin{aligned}
&= \mathsf{eval}^\kappa L >\!\!>\!\!=^\kappa \begin{cases} \mathsf{inl}\, V \mapsto \Delta^\kappa([\![M]\!]^\kappa_{[\![V]\!]^{\mathsf{Val},\kappa}}) \\ \mathsf{inr}\, V \mapsto \Delta^\kappa([\![N]\!]^\kappa_{[\![V]\!]^{\mathsf{Val},\kappa}}) \end{cases} \\
&= \mathsf{eval}^\kappa L >\!\!>\!\!=^\kappa \lambda V.\mathsf{match}\ [\![V]\!]^{\mathsf{Val},\kappa}\ \mathsf{with} \begin{cases} \mathbf{inl}\, v \mapsto (\Delta^\kappa [\![M]\!]^\kappa_v) \\ \mathbf{inr}\, v \mapsto (\Delta^\kappa [\![N]\!]^\kappa_v) \end{cases} \\
&= \left(\mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)(\mathsf{eval}^\kappa L)\right) >\!\!>\!\!=^\kappa \begin{cases} \mathbf{inl}\, v \mapsto \Delta^\kappa([\![M]\!]^\kappa_v) \\ \mathbf{inr}\, v \mapsto \Delta^\kappa([\![N]\!]^\kappa_v) \end{cases} \\
&= [\![L]\!]^\kappa >\!\!>\!\!=^\kappa \begin{cases} \mathbf{inl}\, v \mapsto \Delta^\kappa([\![M]\!]^\kappa_v) \\ \mathbf{inr}\, v \mapsto \Delta^\kappa([\![N]\!]^\kappa_v) \end{cases} \\
&= [\![\mathsf{case}(L, x.M, y.N)]\!]^\kappa
\end{aligned}
$$

Case: $M = \mathsf{fold}\, M'$

$$
\begin{aligned}
&\mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{eval}^\kappa(\mathsf{fold}\, M')\right) \\
&= \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{D}^\kappa((\mathsf{fold}))\left(\mathsf{eval}^\kappa(M')\right)\right) \\
&= \mathsf{D}^\kappa\left([\![\mathsf{fold}\, -]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{eval}^\kappa(M')\right) \\
&= \mathsf{D}^\kappa\left(\mathsf{next}^\kappa \circ [\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{eval}^\kappa(M')\right) \\
&= \mathsf{D}^\kappa(\mathsf{next}^\kappa)\left(\mathscr{D}([\![-]\!]^{\mathsf{Val},\kappa})\left(\mathsf{eval}^\kappa(M')\right)\right) \\
&= \mathsf{D}^\kappa(\mathsf{next}^\kappa)\left([\![M']\!]^\kappa\right) \\
&= [\![\mathsf{fold}\, M']\!]^\kappa
\end{aligned}
$$

Case: $M = \mathsf{unfold}\, M'$

Note that for $V' : \mathsf{Val}_{\tau[\mu \mathsf{X}.\tau/\mathsf{X}]}$ we have that

$$
\triangleright(\alpha : \kappa).\left([\![\mathsf{fold}\, V']\!]^{\mathsf{Val},\kappa}[\alpha] = [\![V']\!]^{\mathsf{Val},\kappa}\right)
$$

and thus we get

$$
\begin{aligned}
&\mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{eval}^\kappa(\mathsf{unfold}\, M')\right) \\
&= \mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\, V').\Delta^\kappa(\delta^\kappa(V'))\right) \\
&= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\, V').\mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\Delta^\kappa(\delta^\kappa(V'))\right) \\
&= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\, V').\mathsf{D}^\kappa\left([\![-]\!]^{\mathsf{Val},\kappa}\right)\left(\Delta^\kappa(\delta^\kappa([\![V']\!]^{\mathsf{Val},\kappa}))\right) \\
&= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\, V').(\mathsf{step}^\kappa(\lambda(\alpha : \kappa).(\delta^\kappa([\![V']\!]^{\mathsf{Val},\kappa})))) \\
&= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda(\mathsf{fold}\, V').(\mathsf{step}^\kappa(\lambda(\alpha : \kappa).(\delta^\kappa([\![\mathsf{fold}\, V']\!]^{\mathsf{Val},\kappa}[\alpha])))) \\
&= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa \lambda V.(\mathsf{step}^\kappa(\lambda(\alpha : \kappa).(\delta^\kappa([\![V]\!]^{\mathsf{Val},\kappa}[\alpha]))))
\end{aligned}
$$

$$
\begin{aligned}
&= \mathsf{eval}^\kappa(M') >\!\!>\!\!=^\kappa (\lambda v.\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\delta^\kappa(v\,[\alpha]))) \circ [\![-]\!]^{\mathsf{Val},\kappa} \\
&= (\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\,(\mathsf{eval}^\kappa(M'))) >\!\!>\!\!=^\kappa \lambda v.\mathsf{step}^\kappa(\lambda(\alpha:\kappa).\delta^\kappa(v\,[\alpha])) \\
&= [\![\mathsf{unfold}\,M']\!]^\kappa
\end{aligned}
$$

Case: $M = \mathsf{choice}^p(N_1, N_2)$

$$
\begin{aligned}
&\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\,(\mathsf{eval}^\kappa(\mathsf{choice}^p(N_1,N_2))) \\
&= \mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\,\big(\mathsf{eval}^\kappa(N_1) \oplus_p^\kappa \mathsf{eval}^\kappa(N_2)\big) \\
&= \Big(\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\,(\mathsf{eval}^\kappa(N_1))\Big) \oplus_p^\kappa \Big(\mathsf{D}^\kappa\Big([\![-]\!]^{\mathsf{Val},\kappa}\Big)\,(\mathsf{eval}^\kappa(N_2))\Big) \\
&= [\![N_1]\!]^\kappa \oplus_p^\kappa [\![N_2]\!]^\kappa \\
&= [\![\mathsf{choice}^p(N_1,N_2)]\!]^\kappa
\end{aligned}
$$

□

Proof of equation (3.7). Note that f is a value by assumption and Y as well as e_f are values by definition. Let $\rho \triangleq x \mapsto v$, and write Δ^κ for $\mathsf{step}^\kappa \circ \mathsf{next}^\kappa$.

$$
\begin{aligned}
&[\![\mathsf{Y}(f)\,x]\!]^\kappa_\rho \\
&= ([\![\mathsf{Y}]\!]^\kappa_\rho \cdot [\![f]\!]^\kappa_\rho) \cdot [\![x]\!]^\kappa_\rho \\
&= \Big((\Delta^\kappa)[\![\mathsf{lam}\ z.e_f(\mathsf{fold}\,(e_f))z]\!]^\kappa_\rho\Big) \cdot [\![x]\!]^\kappa_\rho \\
&= (\Delta^\kappa)^2\,\Big([\![(e_f(\mathsf{fold}\,(e_f)))z]\!]^\kappa_{\rho,z\mapsto [\![x]\!]^{\mathsf{Val},\kappa}_\rho}\Big) \\
&= (\Delta^\kappa)^2\,\Big([\![e_f(\mathsf{fold}\,(e_f))]\!]^\kappa_\rho \cdot [\![x]\!]^\kappa_\rho\Big) \\
&= (\Delta^\kappa)^2\,\Big(\Big([\![e_f]\!]^\kappa_\rho \cdot [\![\mathsf{fold}\,(e_f)]\!]^\kappa_\rho\Big) \cdot [\![x]\!]^\kappa_\rho\Big) \\
&= (\Delta^\kappa)^3\begin{pmatrix}[\![\mathsf{let}\ y' = \mathsf{unfold}\,(\mathsf{fold}\,e_f)\ \mathsf{in}\ f(\mathsf{lam}\ x.(y'(\mathsf{fold}\,e_f))x)]\!]^\kappa_\rho \\ \cdot [\![x]\!]^\kappa_\rho\end{pmatrix} \\
&= (\Delta^\kappa)^3\begin{pmatrix}\Big([\![\mathsf{lam}\ y'.f(\mathsf{lam}\ x.(y'(\mathsf{fold}\,e_f))x)]\!]^\kappa_\rho \cdot [\![\mathsf{unfold}\,(\mathsf{fold}\,e_f)]\!]^\kappa_\rho\Big) \\ \cdot\ [\![x]\!]^\kappa_\rho\end{pmatrix} \\
&= (\Delta^\kappa)^3\,\Big(\Big((\Delta^\kappa)^2[\![(f(\mathsf{lam}\ x.(y'(\mathsf{fold}\,e_f))x))]\!]^\kappa_{\rho,y'\mapsto [\![e_f]\!]^{\mathsf{Val},\kappa}_\rho}\Big) \cdot [\![x]\!]^\kappa_\rho\Big) \\
&= (\Delta^\kappa)^3\,\Big(\Big((\Delta^\kappa)^2\,\Big([\![f]\!]^\kappa_\rho \cdot [\![\mathsf{lam}\ x.(e_f(\mathsf{fold}\,e_f))x]\!]^\kappa_\rho\Big)\Big) \cdot [\![x]\!]^\kappa_\rho\Big) \\
&= (\Delta^\kappa)^3\,\Big(\Big(\Delta^\kappa([\![f]\!]^\kappa_\rho \cdot [\![\mathsf{Y}(f)]\!]^\kappa_\rho)\Big) \cdot [\![x]\!]^\kappa_\rho\Big) \\
&= (\Delta^\kappa)^4\,\Big(\Big([\![f(\mathsf{Y}(f))]\!]^\kappa_\rho\Big) \cdot [\![x]\!]^\kappa_\rho\Big) \\
&= (\Delta^\kappa)^4\,\Big([\![f(\mathsf{Y}(f))\,x]\!]^\kappa_\rho\Big)
\end{aligned}
$$

□

Section 3.7

Proof of Lemma 3.7.3. The proof is organised as follows: We first show how to apply [72, Theorem 4.3] to encode a coinductive type as the one described in Lemma 3.7.3. Then we show that the encoding is the same as $\overline{\mathscr{R}}$.

Recall first the indexed version of [72, Theorem 4.3]: If

$$F : (I \to \mathsf{U}) \to (I \to \mathsf{U})$$

is a functor in the naive sense, such that for any $X : \forall\kappa.(I \to \mathsf{U})$, the canonical map

$$F(\forall\kappa.X) \to \forall\kappa.F(X\,[\kappa])$$

is an equivalence, then a final coalgebra for F can be encoded by first solving $F(\triangleright^\kappa X) \simeq X$ and then defining the final coalgebra as $\forall\kappa.X$. Here $\forall\kappa.X$ means $\lambda(i : I).\forall\kappa.(X\,[\kappa]\,i)$, and $\triangleright^\kappa X$ means $\lambda(i : I).\triangleright^\kappa (X\,i)$

For this application, take $I = \mathsf{D}^\forall(A) \times \mathsf{D}^\forall(A)$, and define $F(P)(\mu,\nu)$ by the clauses of Lemma 3.7.3, i.e., either

1. there exists $\mu' : \mathscr{D}A$ and $\nu' : \mathscr{D}B$ such that $\mu = \mathscr{D}(\mathsf{inl})(\mu')$, $\nu \rightsquigarrow \nu'$ and there exists a coupling $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu,\nu')$, or
2. there exists $\mu' : \mathscr{D}(\mathsf{D}^\forall A)$ such that $\mu = \mathscr{D}(\mathsf{inr})(\mu')$ and $P(\mathsf{run}(\mu'),\nu)$, or
3. there exist $\mu_1 : \mathscr{D}A, \mu_2 : \mathscr{D}(\mathsf{D}^\forall A), \nu_1 : \mathscr{D}B, \nu_2 : \mathsf{D}^\forall B$, and $p : (0,1)$, such that $\mu = \mu_1 \oplus_p \mu_2$ and $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu_1,\nu_1)$, and $P(\mathsf{run}(\mu_2),\nu_2)$.

Given $P : \forall\kappa.(\mathsf{D}^\forall(A) \times \mathsf{D}^\forall(A) \to \mathsf{U})$ we show that $\forall\kappa.F(P\,[\kappa]) \simeq F(\forall\kappa.P)$, the right hand side of this is the statement that either

1. there exist $\mu' : \mathscr{D}A$ and $\nu' : \mathscr{D}B$ such that $\mu = \mathscr{D}(\mathsf{inl})(\mu')$, $\nu \rightsquigarrow \nu'$ and there exists a coupling $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu,\nu')$, or
2. there exists $\mu' : \mathscr{D}(\mathsf{D}^\forall A)$ such that $\mu = \mathscr{D}(\mathsf{inr})(\mu')$ and $\forall\kappa.(P\,[\kappa])(\mathsf{run}(\mu'),\nu)$, or
3. there exist $\mu_1 : \mathscr{D}A, \mu_2 : \mathscr{D}(\mathsf{D}^\forall A), \nu_1 : \mathscr{D}B, \nu_2 : \mathsf{D}^\forall B$, and $p : (0,1)$, such that $\mu = \mu_1 \oplus_p \mu_2$ and $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu_1,\nu_1)$, and $\forall\kappa.(P\,[\kappa])(\mathsf{run}(\mu_2),\nu_2)$.

To prove this, recall the following facts from [72]: $\forall\kappa.(-)$ commutes with sums, products, Σ and propositional truncation. In particular, it commutes with existential quantification over clock irrelevant types. Recall also that our assumption of a clock

constant makes all propositions clock irrelevant. In particular, in $\forall\kappa.F(P\,[\kappa])$ we can commute $\forall\kappa$ first over the case split, then remove it from the case of (1) because this is a proposition not referring to P, and so κ will not appear in it. In (2), we can commute $\forall\kappa$ over first the existential quantification over the clock irrelevant type $\mathscr{D}(\mathsf{D}^{\forall}A)$, then over the product and use that $\mu = \mathscr{D}(\mathsf{inr})(\mu')$ is a proposition, arriving at condition (2) of $F(\forall\kappa.P)$. The case of (3) is similar.

The conclusion from this is that the final coalgebra mentioned in Lemma 3.7.3 can be encoded as $S(\mu,\nu) \triangleq \forall\kappa.S^{\kappa}(\mu,\nu)$, where S^{κ} is the guarded recursive relation satisfying $S^{\kappa}(\mu,\nu)$ iff either

1. there exists $\mu' : \mathscr{D}A$ and $\nu' : \mathscr{D}B$ such that $\mu = \mathscr{D}(\mathsf{inl})(\mu')$, $\nu \rightsquigarrow \nu'$ and there exists a coupling $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu,\nu')$, or

2. there exists $\mu' : \mathscr{D}(\mathsf{D}^{\forall}A)$ such that $\mu = \mathscr{D}(\mathsf{inr})(\mu')$ and $\triangleright^{\kappa}(S^{\kappa}(\mathsf{run}(\mu'),\nu))$, or

3. there exist $\mu_1 : \mathscr{D}A, \mu_2 : \mathscr{D}(\mathsf{D}^{\forall}A), \nu_1 : \mathscr{D}B, \nu_2 : \mathsf{D}^{\forall}B$, and $p : (0,1)$, such that $\mu = \mu_1 \oplus_p \mu_2$ and $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu_1,\nu_1)$, and $\triangleright^{\kappa}(S^{\kappa}(\mathsf{run}(\mu_2),\nu_2))$.

It remains to show that $S^{\kappa}(\mu,\nu)$ is equivalent to $\overline{\mathscr{R}}^{\kappa}(\mu\,[\kappa],\nu)$, which we prove by guarded recursion. First recall that $(\mu\,[\kappa])\,\overline{\mathscr{R}}^{\kappa}\,\nu$ iff either

1. there exist $\mu' : \mathscr{D}A$ and $\nu' : \mathscr{D}B$ such that $(\mu\,[\kappa]) = \mathscr{D}(\mathsf{inl})(\mu')$, and $\nu \rightsquigarrow \nu'$ and there exists a coupling $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu',\nu')$, or

2. there exists $\mu' : \mathscr{D}(\triangleright^{\kappa}\mathsf{D}^{\kappa}A)$ such that $\mu\,[\kappa] = \mathscr{D}(\mathsf{inr})(\mu')$ and $\triangleright^{\kappa}(\overline{\mathscr{R}}^{\kappa}((\zeta^{\kappa}(\mu'))[\alpha]),\nu))$, or

3. there exist $\mu_1 : \mathscr{D}A, \mu_2 : \mathscr{D}(\triangleright^{\kappa}\mathsf{D}^{\kappa}A), \nu_1 : \mathscr{D}B, \nu_2 : \mathsf{D}^{\forall}B$, and $p : (0,1)$, such that $\mu = \mu_1 \oplus_p \mu_2$ and $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu_1,\nu_1)$, and $\triangleright^{\kappa}(\alpha : \kappa)(\zeta^{\kappa}(\mu_2)\,[\alpha]\,\overline{\mathscr{R}}^{\kappa}\,\nu_2)$.

Proving $S^{\kappa}(\mu,\nu)$ implies $\overline{\mathscr{R}}^{\kappa}(\mu\,[\kappa],\nu)$ is easy. For example, in the case of (3) of $S^{\kappa}(\mu,\nu)$, to prove $\overline{\mathscr{R}}^{\kappa}(\mu\,[\kappa],\nu)$ we can reuse μ_1,ν_1, and ν_2, and take μ_2 to be $\mu_2' = \mathscr{D}(\mathsf{next}^{\kappa}\circ\mathsf{ev}_{\kappa})(\mu_2)$. Easy calculations then show that $\mu\,[\kappa] = \mu_1 \oplus_p \mu_2'$ and $\zeta^{\kappa}(\mu_2') = \mathsf{next}^{\kappa}(\mathsf{run}(\mu_2)\,[\kappa])$ so that $\triangleright^{\kappa}(\alpha : \kappa)(\zeta^{\kappa}(\mu_2')\,[\alpha]\,\overline{\mathscr{R}}^{\kappa}\,\nu_2)$ follows from $\triangleright^{\kappa}(S^{\kappa}(\mathsf{run}(\mu_2),\nu_2))$ by guarded recursion.

In the other direction we just show that (3) of $(\mu\,[\kappa])\,\overline{\mathscr{R}}^{\kappa}\,\nu$ implies (3) of $S^{\kappa}(\mu,\nu)$. So suppose the former, and apply the equivalence of Theorem 3.3.7 to μ. By uniqueness of $\mu\,[\kappa] = \mu_1 \oplus_p \mu_2$ it must be the case that also $\mu = \mu_1 \oplus_p \mu_2'$ for some μ_2' such that $\mu_2' = \mathscr{D}(\mathsf{next}^{\kappa}\circ\mathsf{ev}_{\kappa})(\mu_2)$. Now, as before $\zeta^{\kappa}(\mu_2) = \mathsf{next}^{\kappa}(\mathsf{run}(\mu_2')\,[\kappa])$ so that $\triangleright^{\kappa}(S^{\kappa}(\mathsf{run}(\mu_2'),\nu_2))$ follows from $\triangleright^{\kappa}(\alpha : \kappa)(\zeta^{\kappa}(\mu_2)\,[\alpha]\,\overline{\mathscr{R}}^{\kappa}\,\nu_2)$ by guarded recursion. □

Proof of Lemma 3.7.6. For the first statement, notice that $(\mathsf{step}^{\kappa}\mu_1) \oplus_p^{\kappa} (\mathsf{step}^{\kappa}\mu_2) = \mathscr{D}(\mathsf{inr})(\delta(\mu_1) \oplus_p \delta(\mu_2))$, and $\mathsf{step}^{\kappa}(\lambda(\alpha{:}\kappa).(\mu_1\,[\alpha]) \oplus_p^{\kappa} (\mu_2\,[\alpha])) = \mathscr{D}(\mathsf{inr})(\delta(\lambda(\alpha{:}$

$\kappa).(\mu_1\,[\alpha]) \oplus_p^\kappa (\mu_2\,[\alpha]))))$, where both right hands of these equations are of form $\mathscr{D}(\mathsf{inr})(\mu')$ with $\mu' : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$. Therefore, by Definition 3.7.2(2) we have:

$$\begin{aligned}
&(\mathsf{step}^\kappa \mu_1) \oplus_p^\kappa (\mathsf{step}^\kappa \mu_2)\ \overline{\mathscr{R}}^\kappa\ \nu \\
\Leftrightarrow\ &\triangleright(\alpha:\kappa).((((\zeta^\kappa((\delta(\mu_1)) \oplus_p (\delta(\mu_2))))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu) \\
=\ &\triangleright(\alpha:\kappa).(((\mu_1\,[\alpha]) \oplus_p^\kappa (\mu_2\,[\alpha]))\ \overline{\mathscr{R}}^\kappa\ \nu) \\
=\ &\triangleright(\alpha:\kappa).((((\zeta^\kappa(\delta(\lambda(\beta:\kappa).(\mu_1\,[\beta]) \oplus_p^\kappa (\mu_2\,[\beta])))))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu) \\
\Leftrightarrow\ &\mathsf{step}^\kappa(\lambda(\beta:\kappa).(\mu_1\,[\beta]) \oplus_p^\kappa (\mu_2\,[\beta]))\ \overline{\mathscr{R}}^\kappa\ \nu.
\end{aligned}$$

The second statement follows from Lemma 3.7.5 since

$$(\mathsf{step}^\forall(\nu_1)) \oplus_p^\forall (\mathsf{step}^\forall(\nu_2)) \rightsquigarrow \nu_1 \oplus_p^\forall \nu_2$$

and $\mathsf{step}^\forall(\nu_1 \oplus_p^\forall \nu_2) \rightsquigarrow \nu_1 \oplus_p^\forall \nu_2$. □

Proof of Lemma 3.7.7. We analyse the different possibilities for $\mu_1\ \overline{\mathscr{R}}^\kappa\ \nu_1$ and $\mu_2\ \overline{\mathscr{R}}^\kappa\ \nu_2$.

- If both $\mu_1 : \mathscr{D}A$ and $\mu_2 : \mathscr{D}A$, then there exist ν_1', ν_2' such that $\nu_1 \rightsquigarrow \nu_1'$ and $\nu_2 \rightsquigarrow \nu_2'$, and there exist $\rho_1 : \mathsf{Cpl}_{\mathscr{R}}(\mu_1', \nu_1'), \rho_2 : \mathsf{Cpl}_{\mathscr{R}}(\mu_2', \nu_2')$. Then:

  $$\begin{aligned}
  &\nu_1 \oplus_p^\forall \nu_2 \rightsquigarrow \nu_1' \oplus_p^\forall \nu_2' \\
  &\rho_1 \oplus_p \rho_2 : \mathsf{Cpl}_{\mathscr{R}}(\mu_1' \oplus_p \mu_2', \nu_1' \oplus_p \nu_2')
  \end{aligned}$$

  And hence: $(\mu_1 \oplus_p^\kappa \mu_2)\ \overline{\mathscr{R}}^\kappa\ (\nu_1 \oplus_p^\forall \nu_2)$.

- If $\mu_1 : \mathscr{D}A$ and $\mu_2 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$, then taking $\rho = \rho_1 : \mathsf{Cpl}_{\mathscr{R}}(\mu_1, \nu_1)$ that we get from $\mu_1\ \overline{\mathscr{R}}^\kappa\ \nu_1$ makes $\mu_1 \oplus_p^\kappa \mu_2$ satisfy the third option for proving $(\mu_1 \oplus_p^\kappa \mu_2)\ \overline{\mathscr{R}}^\kappa\ (\nu_1 \oplus_p^\forall \nu_2)$.

- A similar reasoning shows the case for $\mu_1 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ and $\mu_2 : \mathscr{D}A$, after applying the commutativity axiom for convex algebras.

- If $\mu_1 : \mathscr{D}A$ and for μ_2 there exist $\mu_2' : \mathscr{D}A, \mu_2'' : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ such that $\mu_2 = \mu_2' \oplus_q \mu_2''$, then there exist $\nu_1', \nu_2' : \mathscr{D}B, \nu_2'' : \mathsf{D}^\forall B$ such that $\nu_1 \rightsquigarrow \nu_1'$ and $\nu_2 \rightsquigarrow \nu_2' \oplus_q \nu_2''$. There also exist $\rho_1 : \mathsf{Cpl}_{\mathscr{R}}(\mu_1, \nu_1')$ and $\rho_2 : \mathsf{Cpl}_{\mathscr{R}}(\mu_2', \nu_2')$. Then using associativity we can compute probabilities p', q' such that:

  $$\mu_1 \oplus_p (\mu_2' \oplus_q \mu_2'') = (\mu_1 \oplus_{p'} \mu_2') \oplus_{q'} \mu_2''$$

  Then $\nu_1 \oplus_p^\forall \nu_2 \rightsquigarrow (\nu_1' \oplus_{p'} \nu_2') \oplus_{q'} \nu_2''$, and

  $$\rho_1 \oplus_{p'} \rho_2 : \mathsf{Cpl}_{\mathscr{R}}(\mu_1 \oplus_{p'} \mu_2', \nu_1' \oplus_{p'} \nu_2')$$

  and from our assumption for μ_2 we still have that:

  $$\triangleright^\kappa(\alpha:\kappa)\ (\zeta^\kappa(\mu_2'')\,[\alpha]\ \overline{\mathscr{R}}^\kappa\ \nu_2'').$$

  Hence, $(\mu_1 \oplus_p^\kappa \mu_2)\ \overline{\mathscr{R}}^\kappa\ (\nu_1 \oplus_p^\forall \nu_2)$ holds via option 3.

- A similar line of reasoning shows the case for $\mu_2 : \mathscr{D}A$ and for μ_1 there exist $\mu_1' : \mathscr{D}A, \mu_1'' : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ such that $\mu_1 = \mu_1' \oplus_q \mu_1''$, except that we now need commutativity as well as associativity to show that there exist probabilities p', q' such that:

$$(\mu_1' \oplus_q \mu_1'') \oplus_p \mu_2 = (\mu_1' \oplus_{q'} \mu_2) \oplus_{p'} \mu_1''$$

- If both $\mu_1 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ and $\mu_2 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$, then:

$$\triangleright^\kappa (\alpha : \kappa)\ (((\zeta^\kappa(\mu_1))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu_1)$$
$$\triangleright^\kappa (\alpha : \kappa)\ (((\zeta^\kappa(\mu_2))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu_2)$$

  It follows from guarded recursion and the fact that

$$\zeta^\kappa(\mu_1 \oplus_p \mu_2) = \lambda(\alpha{:}\kappa).\zeta^\kappa(\mu_1)\,[\alpha] \oplus_p^\kappa \zeta^\kappa(\mu_2)\,[\alpha],$$

  that then also:

$$\triangleright^\kappa(\alpha : \kappa)\ (((\zeta^\kappa(\mu_1 \oplus_p^\kappa \mu_2))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu_1 \oplus_p^\forall \nu_2),$$

  and hence the second option for $(\mu_1 \oplus_p^\kappa \mu_2)\ \overline{\mathscr{R}}^\kappa\ (\nu_1 \oplus_p^\forall \nu_2)$ is satisfied.

- If $\mu_1 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ and for μ_2 there exist $\mu_2' : \mathscr{D}A, \mu_2'' : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ such that $\mu_2 = \mu_2' \oplus_q \mu_2''$, then there exist $\nu_2' : \mathscr{D}B, \nu_2'' : \mathsf{D}^\forall B$ such that $\nu_2 \rightsquigarrow \nu_2' \oplus_q \nu_2''$. There also exist $\rho_2 : \mathsf{Cpl}_{\mathscr{R}}(\mu_2', \nu_2')$. We again compute probabilities p', q' such that:

$$\mu_1 \oplus_p (\mu_2' \oplus_q \mu_2'') = \mu_2' \oplus_{p'} (\mu_1 \oplus_{q'} \mu_2'')$$

  Then ρ_2 still serves as the needed coupling. We also have:

$$\triangleright^\kappa (\alpha : \kappa)\ (((\zeta^\kappa(\mu_1))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu_1)$$
$$\triangleright^\kappa (\alpha : \kappa)\ (((\zeta^\kappa(\mu_2''))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu_2'')$$

  Then by guarded recursion and the fact that $\zeta^\kappa(\mu_1 \oplus_{q'} \mu_2'') = \lambda(\alpha{:}\kappa).\zeta^\kappa(\mu_1)\,[\alpha] \oplus_{q'}^\kappa \zeta^\kappa(\mu_2'')\,[\alpha]$, we also have:

$$\triangleright^\kappa(\alpha : \kappa)\ (((\zeta^\kappa(\mu_1 \oplus_{q'}^\kappa \mu_2''))[\alpha])\ \overline{\mathscr{R}}^\kappa\ \nu_1 \oplus_{q'}^\forall \nu_2''),$$

  which is the last requirement for $(\mu_1 \oplus_p^\kappa \mu_2)\ \overline{\mathscr{R}}^\kappa\ (\nu_1 \oplus_p^\forall \nu_2)$ to hold via the third condition.

- A similar line of reasoning shows the case for $\mu_2 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ and for μ_1 there exist $\mu_1' : \mathscr{D}A, \mu_1'' : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ such that $\mu_1 = \mu_1' \oplus_q \mu_1''$.

- In the last case, we get:

$$\begin{array}{lll} \mu_1 = \mu_1' \oplus_q \mu_1'' & \nu_1 \rightsquigarrow \nu_1' \oplus_q \nu_1'' & \triangleright^\kappa(\alpha:\kappa)\ (\zeta^\kappa(\mu_1'')\,[\alpha]\,\overline{\mathscr{R}}^\kappa\,\nu_1'') \\ \mu_2 = \mu_2' \oplus_r \mu_2'' & \nu_2 \rightsquigarrow \nu_2' \oplus_r \nu_2'' & \triangleright^\kappa(\alpha:\kappa)\ (\zeta^\kappa(\mu_2'')\,[\alpha]\,\overline{\mathscr{R}}^\kappa\,\nu_2'') \end{array}$$

  and $\rho_1 : \mathsf{Cpl}_{\mathscr{R}}(\mu_1', \nu_1'), \rho_2 : \mathsf{Cpl}_{\mathscr{R}}(\mu_2', \nu_2')$.

  From the axioms of convex algebras we now compute probabilities p', q', r' such that:

$$\begin{aligned} (\mu_1' \oplus_q \mu_1'') \oplus_p (\mu_2' \oplus_r \mu_2'') &= (\mu_1' \oplus_{q'} \mu_2') \oplus_{p'} (\mu_1'' \oplus_{r'} \mu_2'') \\ (\nu_1' \oplus_q \nu_1'') \oplus_p (\nu_2' \oplus_r \nu_2'') &= (\nu_1' \oplus_{q'} \nu_2') \oplus_{p'} (\nu_1'' \oplus_{r'} \nu_2'') \end{aligned}$$

  Then $\nu_1 \oplus_p^\forall \nu_2 \rightsquigarrow (\nu_1' \oplus_{q'} \nu_2') \oplus_{p'} (\nu_1'' \oplus_{r'} \nu_2'')$ and since

$$\rho_1 \oplus_{q'} \rho_2 : \mathsf{Cpl}_{\mathscr{R}}(\mu_1' \oplus_{q'} \mu_1'', \nu_1' \oplus_{q'} \nu_1'')$$

  it just remains show that $\triangleright^\kappa(\alpha:\kappa)\ (\zeta^\kappa(\mu_1'' \oplus_{r'} \mu_2'')\,[\alpha]\,\overline{\mathscr{R}}^\kappa\,\nu_1'')$. This follows from guarded recursion, since

$$\zeta^\kappa(\mu_1'' \oplus_{r'} \mu_2'') = \lambda(\alpha:\kappa).\zeta^\kappa(\mu_1'')\,[\alpha] \oplus_{r'}^\kappa \zeta^\kappa(\mu_2'')\,[\alpha].$$

□

Proof of Lemma 3.7.8. The first statement is trivial statement is trivial by taking $\rho = \delta((a,b))$. For the second statement, suppose that $\mu\ \overline{\mathscr{R}}^\kappa\ \nu$. We consider each of the three cases.

- If $\mu : \mathscr{D}A$, $\nu \rightsquigarrow \nu'$, and $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu, \nu')$, then we proceed by induction on ρ: If $\rho = \delta((a,b))$ for some $a : A$ and $b : B$ such that $a\,\mathscr{R}\,b$, then $\overline{f}(\mu) = \overline{f}(\delta^\kappa a) = f(a)$ and $\overline{g}(\nu') = \overline{g}(\delta^\kappa b) = g(b)$ are related in $\overline{\mathscr{S}}^\kappa$ by assumption.

  If $\rho = \rho_1 \oplus_q \rho_2$, let $\mu_i \triangleq \mathscr{D}(\mathsf{pr}_1)(\rho_i)$ and $\nu_i \triangleq \mathscr{D}(\mathsf{pr}_2)(\rho_i)$ for $i = 1,2$. By induction $\overline{f}(\mu_i)\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu_i)$ and since

$$\begin{aligned} \overline{f}(\mu) = \overline{f}(\mu_1 \oplus_p^\kappa \mu_2) &= \overline{f}(\mu_1) \oplus_p^\kappa \overline{f}(\mu_2) \\ \overline{g}(\nu) = \overline{g}(\nu_1 \oplus_p^\forall \nu_2) &= \overline{g}(\nu_1) \oplus_p^\forall \overline{g}(\nu_2) \end{aligned}$$

  the case follows from Lemma 3.7.7.

- If $\mu : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ and $\triangleright^\kappa(\alpha:\kappa)\ (((\zeta^\kappa(\mu))[\alpha])\,\overline{\mathscr{R}}^\kappa\,\nu)$, note that $\triangleright^\kappa(\alpha:\kappa)\ (\zeta^\kappa(\overline{f}(\mu))[\alpha]) = \overline{f}(\zeta^\kappa(\mu)[\alpha])$. We show this by induction on μ: If $\mu : \delta(\mu')$, then:

$$\begin{aligned} &\zeta^\kappa(\overline{f}(\delta(\mu')))[\alpha] \\ &= (\zeta^\kappa(\delta(\lambda(\beta:\kappa).(\overline{f}(\mu'\,[\beta])))))[\alpha] \\ &= (\lambda(\beta:\kappa).\overline{f}(\mu'\,[\beta]))[\alpha] \end{aligned}$$

$$
\begin{aligned}
&= \overline{f}(\mu'[\alpha]) \\
&= \overline{f}(\zeta^\kappa(\delta(\mu'))[\alpha]).
\end{aligned}
$$

If $\mu = \mu' \oplus_q \mu''$, then:

$$
\begin{aligned}
&(\zeta^\kappa(\overline{f}(\mu' \oplus_q \mu'')))[\alpha] \\
&= (\zeta^\kappa(\overline{f}(\mu') \oplus_q \overline{f}(\mu')))[\alpha] \\
&= \lambda(\beta:\kappa).((\zeta^\kappa(\overline{f}(\mu')))[\beta]) \oplus_q ((\zeta^\kappa(\overline{f}(\mu')))[\beta])[\alpha] \\
&= (\zeta^\kappa(\overline{f}(\mu')))[\alpha] \oplus_q (\zeta^\kappa(\overline{f}(\mu')))[\alpha] \\
&= (\overline{f}(\zeta^\kappa\mu')[\alpha]) \oplus_q (\overline{f}(\zeta^\kappa\mu'')[\alpha]) \\
&= \overline{f}(((\zeta^\kappa\mu')[\alpha]) \oplus_q ((\zeta^\kappa\mu'')[\alpha])) \\
&= \overline{f}(\zeta^\kappa(((\mu')[\alpha]) \oplus_q ((\mu'')[\alpha])))
\end{aligned}
$$

Hence we may conclude by guarded recursion that

$$\triangleright^\kappa(\alpha:\kappa)\ ((\zeta^\kappa(\overline{f}(\mu))[\alpha])\,\overline{\mathscr{S}}^\kappa\,(\overline{g}(\nu))),$$

which proves that $\overline{f}(\mu)\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu)$ via the second condition.

- If there are $\mu_1 : \mathscr{D}A, \mu_2 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$, $\nu_1 : \mathscr{D}B, \nu_2 : \mathsf{D}^\forall B$, and $\rho : \mathsf{Cpl}_{\mathscr{R}}(\mu, \nu_1)$, such that $\mu = \mu_1 \oplus_p \mu_2$ and $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and $\triangleright^\kappa(\alpha:\kappa)\ (\zeta^\kappa(\mu)\,[\alpha]\,\overline{\mathscr{R}}^\kappa\,\nu_2)$, then since $\overline{f}(\mu) = \overline{f}(\mu_1) \oplus_p \overline{f}(\mu_2)$ and by Lemma 3.4.7 $\overline{g}(\nu) \rightsquigarrow \overline{g}(\nu_1) \oplus_p \overline{g}(\nu_2)$, it suffices by Lemma 3.7.7 and 3.7.5 to show that $\overline{f}(\mu_1)\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu_1)$ and $\overline{f}(\mu_2)\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu_2)$.

  The first one of these is shown by induction on ρ, similarly to the case of $\mu : \mathscr{D}A$ above.

  To show that $\overline{f}(\mu_2)\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu_2)$, notice that since $\mu_2 : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$, $\overline{f}(\mu_2) : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A')$. Therefore, we only need to show that $\triangleright^\kappa(\alpha:\kappa)\ (((\zeta^\kappa(\overline{f}(\mu_2)))[\alpha])\,\overline{\mathscr{S}}^\kappa\,\overline{g}(\nu_2))$. This follows by guarded recursion, similarly to the case for $\mu : \mathscr{D}(\triangleright^\kappa \mathsf{D}^\kappa A)$ above.

□

Proof of Lemma 3.7.4. By induction on n. For $n = 0$, we do a case analysis on $\mu\ \overline{\mathsf{eq}_1}\ \nu$.

- If $\mu : \mathscr{D}1$ and $\exists \nu' : \mathscr{D}1$ such that $\nu \rightsquigarrow \nu'$, and there exists $\rho : \mathsf{Cpl}_{\mathsf{eq}_1}(\mu, \nu')$, then $\mathsf{PT}_0(\mu) = 1$, and also $\mathsf{PT}_0(\nu') = 1$. Furthermore, by Lemma 3.4.6(3) there is an m such that $\nu' \rightsquigarrow \mathsf{run}^m \nu$. By Lemma 3.4.8 then, we have that $\mathsf{PT}_m(\nu) = \mathsf{PT}_0(\mathsf{run}^m \nu) \geq \mathsf{PT}_0(\nu') = \mathsf{PT}_0(\mu) = 1$, which proves the statement.
- If $\mu : \mathscr{D}(\mathsf{D}^\forall 1)$, then $\mathsf{PT}_0(\mu) = 0$ and the statement is trivially true.
- If there exist $\mu_1 : \mathscr{D}1, \mu_2 : \mathscr{D}(\mathsf{D}^\forall 1)$ such that $\mu = \mu_1 \oplus_p \mu_2$, and there exist $\nu_1 : \mathscr{D}1, \nu_2 : \mathsf{D}^\forall 1$ such that $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathsf{eq}_1}(\mu_1, \nu_1)$, then:

- $\mathsf{PT}_0(\mu) = p$
- By Lemma 3.4.6(3) there is an m such that $\nu_1 \oplus_p \nu_2 \rightsquigarrow \mathsf{run}^m \nu$. Then by Lemma 3.4.8, $\mathsf{PT}_m(\nu) = \mathsf{PT}_0(\mathsf{run}^m \nu) \geq \mathsf{PT}_0(\nu_1 \oplus_p \nu_2)$. In addition, $\mathsf{PT}_0(\nu_1 \oplus_p \nu_2) \geq p$.

Hence, $\mathsf{PT}_0(\mu) \leq \mathsf{PT}_m(\nu)$.

For $n = n' + 1$, we again analyse each case for $\mu \overline{\mathsf{eq}_1} \nu$.

- The case where $\mu : \mathscr{D}1$ is identical to the one above for $n = 0$.
- If $\mu : \mathscr{D}(\mathsf{D}^\forall 1)$ and $\mathsf{run}(\mu) \overline{\mathsf{eq}_1} \nu$ then $\mathsf{PT}_n(\mu) = \mathsf{PT}_{n'}(\mathsf{run}\mu)$. By the induction hypothesis for n', we know that then there is an m such that $\mathsf{PT}_{n'}(\mathsf{run}\mu) \leq \mathsf{PT}_m(\nu)$, and hence also $\mathsf{PT}_n(\mu) \leq \mathsf{PT}_m(\nu)$.
- If there exist $\mu_1 : \mathscr{D}1, \mu_2 : \mathscr{D}(\mathsf{D}^\forall 1)$ such that $\mu = \mu_1 \oplus_p \mu_2$, and there exist $\nu_1 : \mathscr{D}1, \nu_2 : \mathsf{D}^\forall 1$ such that $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$, and there exists a $\rho : \mathsf{Cpl}_{\mathsf{eq}_1}(\mu_1, \nu_1)$, and moreover $\mathsf{run}(\mu_2) \overline{\mathsf{eq}_1} \nu_2$, then:
  - $\mathsf{PT}_n(\mu) = p + (1-p)\mathsf{PT}_{n'}((\mathsf{run}(\mu_2)))$.
  - From the induction hypothesis for n' and the fact that $\mathsf{run}(\mu_2) \overline{\mathsf{eq}_1} \nu_2$, we may conclude that there is an m' such that $\mathsf{PT}_{n'}(\mathsf{run}(\mu_2)) \leq \mathsf{PT}_{m'}(\nu_2)$.
  - From $\nu \rightsquigarrow \nu_1 \oplus_p \nu_2$ and Lemma 3.4.6(3), we know that there is an m such that $\nu_1 \oplus_p \nu_2 \rightsquigarrow \mathsf{run}^m \nu$.

  Then by Lemma 3.4.8 we have:

$$\begin{aligned}
\mathsf{PT}_{m+m'}(\nu) &= \mathsf{PT}_{m'}(\mathsf{run}^m \nu) \\
&\geq \mathsf{PT}_{m'}(\nu_1 \oplus_p \nu_2) \\
&= p + (1-p)\mathsf{PT}_{m'}(\nu_2) \\
&\geq p + (1-p)\mathsf{PT}_{n'}(\mathsf{run}(\mu_2)) \\
&= \mathsf{PT}_n(\mu),
\end{aligned}$$

  which is what we needed to show.

□

Section 3.8

In preparation for the *guarded fundamental theorem* 3.8.2 and *guarded congruence theorem* 3.8.4 we prove several lemmas which show that $\preceq_\sigma^{\kappa,\mathsf{Tm}}$ is compatible with the typing rules in Figure 3.2.

Lemma .0.5. *Let* $f : \sigma \to \tau$ *and* $g : \mathsf{Val}_\sigma \to \mathsf{Val}_\tau$ *as well as* $\mu : \mathsf{D}^\kappa\sigma$ *and* $\nu : \mathsf{D}^\forall(\mathsf{Val}_\sigma)$ *such that* $\mu \preceq^{\kappa,\mathsf{Tm}}_\sigma \nu$. *If for any* $v : \sigma$ *and* $V : \mathsf{Val}_\sigma$ *we have that* $(v \preceq^{\kappa,\mathsf{Val}}_\sigma V) \to (f(v) \preceq^{\kappa,\mathsf{Val}}_\tau g(V))$ *it follows that*

$$(\mu >>=^\kappa \delta^\kappa \circ f) \preceq^{\kappa,\mathsf{Tm}}_\tau (\nu >>= \delta^\forall \circ g).$$

or equivalently

$$\mathsf{D}^\kappa(f)(\mu) \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{D}^\forall(g)(\nu)$$

Proof. This is a direct consequence of 3.7.8. Looking at the requirements, it suffices to prove that $v \preceq^{\kappa,\mathsf{Val}}_\sigma V$ implies $\delta^\kappa(f(v)) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} \delta^\forall(g(V))$. Using 3.7.8, this follows from our assumption that $(v \preceq^{\kappa,\mathsf{Val}}_\sigma V) \to (f(v) \preceq^{\kappa,\mathsf{Val}}_\tau g(V))$. □

Lemma .0.6. *Let* $f : (\sigma_1 \times \sigma_2) \to \sigma_3$ *and* $g : (\mathsf{Val}_{\sigma_1} \times \mathsf{Val}_{\sigma_2}) \to \mathsf{Val}_{\sigma_3}$ *as well as* $\mu \preceq^{\kappa,\mathsf{Tm}}_{\sigma_1} d$ *and* $\nu \preceq^{\kappa,\mathsf{Tm}}_{\sigma_2} e$. *If we have that*

$$(v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1} V) \to (w \preceq^{\kappa,\mathsf{Val}}_{\sigma_2} W) \to (f(v,w) \preceq^{\kappa,\mathsf{Val}}_{\sigma_3} g(V,W))$$

it follows that

$$\begin{aligned}&(\mu >>=^\kappa \lambda v.\nu >>=^\kappa \lambda w.\delta^\kappa(f(v,w)))\\ \preceq^{\kappa,\mathsf{Tm}}_\tau\ &(d >>= \lambda V.e >>= \lambda W.\delta^\forall(g(V,W))).\end{aligned}$$

Proof. We apply 3.7.8 with the functions $\lambda v.\nu >>=^\kappa \lambda w.\delta^\kappa(f(v,w))$ and $\lambda V.e >>= \lambda W.\delta^\forall(g(V,W))$. Since we have that $\mu \preceq^{\kappa,\mathsf{Tm}}_{\sigma_1} d$, it suffices to show that

$$\begin{aligned}(v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1} V) \to ((\nu >>=^\kappa \lambda w.\delta^\kappa(f(v,w)))\\ \preceq^{\kappa,\mathsf{Tm}}_{\sigma_3} (e >>= \lambda W.\delta^\forall(g(V,W))))\end{aligned}$$

Let now $v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1} V$, using .0.5 for the functions $\lambda w.\delta^\kappa(f(v,w))$ and $\lambda W.\delta^\forall(g(V,W))$, it suffices to show that

$$(w \preceq^{\kappa,\mathsf{Val}}_{\sigma_2} W) \to (f(v,w) \preceq^{\kappa,\mathsf{Val}}_{\sigma_3} g(V,W))$$

and the claim follows. □

Corollary .0.7. *Let* $\mathbf{op} \in \{\mathbf{suc}, \mathbf{pred}\}$ *and* $\mu \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} \nu$. *Then*

$$\mathsf{D}^\kappa(\mathbf{op})(\mu) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} \mathsf{D}^\forall(\mathbf{op})(\nu).$$

Proof. This is a direct consequence of .0.5. Note that if $n = m$, then also $\mathbf{op}(n) = \mathbf{op}(m)$. □

The arguments for $\mathbf{inl}$, $\mathbf{inr}$, pr_1 and pr_2 proceed similarly.

Corollary .0.8. *Assume that* $\mu_1 \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{Nat}} \nu_1$, $\mu_2 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \nu_2$ *and* $\mu_3 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \nu_3$. *Then also*

$$\left(\mu_1 >\!\!>\!\!=^{\kappa} \begin{cases} 0 & \mapsto \mu_2 \\ n+1 & \mapsto \mu_3 \end{cases}\right) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \left(\nu_1 >\!\!>\!\!= \begin{cases} \underline{0} & \mapsto \nu_2 \\ \underline{n+1} & \mapsto \nu_3 \end{cases}\right)$$

Proof. We apply the bind lemma for $f = \begin{cases} 0 & \mapsto \mu_2 \\ n+1 & \mapsto \mu_3 \end{cases}$ and $g = \begin{cases} \underline{0} & \mapsto \nu_2 \\ \underline{n+1} & \mapsto \nu_3 \end{cases}$. It thus suffices to show that for any $u \preceq^{\kappa,\mathsf{Val}}_{\mathsf{Nat}} U$ we have that $f(u) \preceq^{\kappa,\mathsf{Tm}}_{\sigma} g(U)$.

We proceed by case distinction on $u : \mathsf{Nat}$.

Case: $u = 0$

Then necessarily also $U = \underline{0}$ and thus $f(u) = \mu_2$ and $g(U) = \nu_2$. But $\mu_2 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \nu_2$ is one of our assumptions.

Case: $u = n+1$

This case is analogous to the previous case. □

Corollary .0.9. *Let* $\mu \preceq^{\kappa,\mathsf{Tm}}_{\sigma} d$ *and* $\nu \preceq^{\kappa,\mathsf{Tm}}_{\tau} e$, *then*

$$\begin{aligned}&(\mu >\!\!>\!\!=^{\kappa} \lambda v.\nu >\!\!>\!\!=^{\kappa} \lambda w.\delta^{\kappa}((v,w))) \\ &\preceq^{\kappa,\mathsf{Tm}}_{\sigma\times\tau} (d >\!\!>\!\!= \lambda V.e >\!\!>\!\!= \lambda W.\delta^{\forall}(\langle V,W\rangle)).\end{aligned}$$

Proof. Using .0.6 it suffices to show that

$$(v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V) \to (w \preceq^{\kappa,\mathsf{Val}}_{\tau} W) \to ((v,w) \preceq^{\kappa,\mathsf{Val}}_{\sigma\times\tau} (V,W))$$

which is immediate by the definition of $(v,w) \preceq^{\kappa,\mathsf{Val}}_{\sigma\times\tau} (V,W)$. □

Lemma .0.10. *Let* $\mu_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} \nu_2$ *and* $\mu_2 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \nu_2$, *then*

$$\mu_1 \cdot \mu_2 \preceq^{\kappa,\mathsf{Tm}}_{\tau} (\nu_1 >\!\!>\!\!= \lambda(\mathsf{lam}\ x.M).\nu_2 >\!\!>\!\!= \lambda W.\mathsf{step}^{\kappa}(\mathsf{eval}(M[W/x]))).$$

Proof. Unfolding the definition we get

$$\begin{aligned}&\left(\begin{array}{l} \mu_1 \cdot \mu_2 \\ \preceq^{\kappa,\mathsf{Tm}}_{\tau} (\nu_1 >\!\!>\!\!= \lambda(\mathsf{lam}\ x.M).\nu_2 >\!\!>\!\!= \lambda W.\mathsf{step}^{\kappa}(\mathsf{eval}(M[W/x]))) \end{array}\right) \\ \leftrightarrow &\left(\begin{array}{l} (\mu_1 >\!\!>\!\!=^{\kappa} \lambda v.\mu_2 >\!\!>\!\!=^{\kappa} \lambda w.\Delta^{\kappa}(v(w))) \\ \preceq^{\kappa,\mathsf{Tm}}_{\tau} (\nu_1 >\!\!>\!\!= \lambda(\mathsf{lam}\ x.M).\nu_2 >\!\!>\!\!= \lambda W.\mathsf{step}^{\kappa}(\mathsf{eval}(M[W/x]))) \end{array}\right)\end{aligned}$$

To show this, we use 3.7.8 on the functions

1. $f_1 \triangleq \lambda v.\mu_2 >\!\!>\!\!=^{\kappa} \lambda w.\Delta^{\kappa}(v(w))$
2. $g_1 \triangleq \lambda(\mathsf{lam}\ x.M).\nu_2 >\!\!>\!\!=^{\kappa} \Lambda W.\mathsf{step}^{\forall}(\mathsf{eval}(M[W/x]))$

and it remains to show that assuming $v \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\ x.M$ we get that

$$(\mu_2 >\!\!>\!\!=^{\kappa} \lambda w.\Delta^{\kappa}(v(w))) \preceq^{\kappa,\mathsf{Tm}}_{\tau} \left(\nu_2 >\!\!>\!\!= \lambda W.\mathsf{step}^{\forall}(\mathsf{eval}(M[W/x]))\right)$$

We proceed by using 3.7.8 once more, this time for the functions

1. $f_2 \triangleq \lambda w.\Delta^\kappa(v(w))$ and
2. $g_2 \triangleq \lambda W.\mathsf{step}^\forall(\mathsf{eval}(M[W/x]))$.

Now it suffices to prove that assuming $(v \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\ x.M)$ and $(w \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} W)$ we get

$$\Delta^\kappa v(w) \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{step}^\forall(\mathsf{eval}(M[W/x]))$$

Observe, that by the definition of $v \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\ x.M$ we immediatly get that $v(w) \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{eval}(M[W/x])$ and thus by Definition 3.7.2 and Lemma 3.7.5 imply the result. □

Lemma .0.11. *Assume* $\mu_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma_1+\sigma_2} \nu_1$ *as well as*

1. $\forall v,V.v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1} V \to (\mathsf{step}^\kappa\lambda(\alpha:\kappa).e_1[\alpha](v)) \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{step}^\forall(\mathsf{eval}(M[V/x]))$
2. $\forall v,V.v \preceq^{\kappa,\mathsf{Val}}_{\sigma_2} V \to (\mathsf{step}^\kappa\lambda(\alpha:\kappa).e_1[\alpha](v)) \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{step}^\forall(\mathsf{eval}(N[V/x]))$

then it follows that

$$\left(\mu_1 >\!\!>\!\!=^\kappa \begin{cases}\mathbf{inl}\,v \mapsto \mathsf{step}^\kappa(\lambda(\alpha:\kappa).e_1[\alpha](v)) \\ \mathbf{inr}\,v \mapsto \mathsf{step}^\kappa(\lambda(\alpha:\kappa).e_2[\alpha](v))\end{cases}\right) \preceq^{\kappa,\mathsf{Tm}}_\tau \left(\nu_1 >\!\!>\!\!= \begin{cases}\mathbf{inl}\,v \mapsto \mathsf{step}^\forall(\mathsf{eval}M[V/x]) \\ \mathbf{inr}\,v \mapsto \mathsf{step}^\forall(\mathsf{eval}N[V/x])\end{cases}\right)$$

Proof. Again, we use 3.7.8 and considering the assumptions of the lemma, it only remains to show that given $w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} W$ we also have

$$\left(\begin{cases}\mathbf{inl}\,v \mapsto \mathsf{step}^\kappa(\lambda(\alpha:\kappa).e_1[\alpha](v)) \\ \mathbf{inr}\,v \mapsto \mathsf{step}^\kappa(\lambda(\alpha:\kappa).e_2[\alpha](v))\end{cases}\right)(w) \preceq^{\kappa,\mathsf{Tm}}_\tau \left(\begin{cases}\mathbf{inl}\,v \mapsto \mathsf{step}^\forall(\mathsf{eval}M[V/x]) \\ \mathbf{inr}\,v \mapsto \mathsf{step}^\forall(\mathsf{eval}N[V/x])\end{cases}\right)(W)$$

This we can show by a case distinction on $w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} W$.

Case: $\mathbf{inl}\,w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} \mathsf{inl}\,W$
In this case it remains to show that

$$\mathsf{step}^\kappa(\lambda(\alpha:\kappa).e_1[\alpha](w)) \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{step}^\forall(\mathsf{eval}M[W/x])$$

which follows from the assumptions.

Case: $\mathbf{inr}\,w \preceq^{\kappa,\mathsf{Val}}_{\sigma_1+\sigma_2} \mathsf{inr}\,W$
In this case it remains to show that

$$\mathsf{step}^\kappa(\lambda(\alpha:\kappa).e_2[\alpha](w)) \preceq^{\kappa,\mathsf{Tm}}_\tau \mathsf{step}^\forall(\mathsf{eval}N[W/x])$$

which again follows from the assumptions.

□

Lemma .0.12. *If* $\mu \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} \nu$ *then also*

$$\mathsf{D}^{\kappa}(\mathsf{next}^{\kappa})(\mu) \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} \mathsf{D}^{\forall}(\mathsf{fold})(\nu).$$

Proof. We apply .0.5 to $f \triangleq \mathsf{next}^{\kappa}$ and $g \triangleq \mathsf{fold}$. It thus suffices to show that for $v \preceq^{\kappa,\mathsf{Val}}_{\tau[\mu X.\tau/X]} V$ we have that $\mathsf{next}^{\kappa} v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} \mathsf{fold} V$. This is however immediate, since

$$\begin{aligned}\mathsf{next}^{\kappa} v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} \mathsf{fold} V &\triangleq (\triangleright(\alpha:\kappa).(\mathsf{next}^{\kappa} v)[\alpha] \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} V)\\ &\leftrightarrow \triangleright(\alpha:\kappa).(v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} V)\end{aligned}$$

□

Lemma .0.13. *If* $\mu \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} \nu$ *then*

$$\begin{aligned}&(\mu >\!\!>\!\!=^{\kappa} \lambda v.\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\delta^{\kappa}(v[\alpha])))\\ &\preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} (\nu >\!\!>\!\!=^{\kappa} \lambda(\mathsf{fold} V).\mathsf{step}^{\forall}(\delta^{\forall}(V)))\end{aligned}$$

Proof. We apply 3.7.8 with the functions $\lambda v.\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).v[\alpha])$ and $\lambda(\mathsf{fold} V).\mathsf{step}^{\forall}(\delta^{\forall}(V))$. It thereby suffices to show that if $v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} (\mathsf{fold} V)$ then also

$$\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\delta^{\kappa}(v[\alpha])) \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} \mathsf{step}^{\forall}(\delta^{\forall}(V)).$$

Note that $v \preceq^{\kappa,\mathsf{Val}}_{\mu X.\tau} (\mathsf{fold} V)$ is equivalent to $\triangleright(\alpha:\kappa).(v[\alpha] \preceq^{\kappa,\mathsf{Val}}_{\tau[\mu X.\tau/X]} V)$. Then $\triangleright(\alpha:\kappa).(\delta^{\kappa}(v[\alpha]) \preceq^{\kappa,\mathsf{Val}}_{\tau[\mu X.\tau/X]} \delta^{\forall}(V))$ by 3.7.8 and it follows by 3.7.2 and 3.7.5 that this is equivalent to

$$\mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).\delta^{\kappa}(v[\alpha])) \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} \mathsf{step}^{\forall}(\delta^{\forall}(V)).$$

This concludes the claim. □

Lemma .0.14. *If* $\mu \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \nu$ *and* $\mu_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \nu_1$ *then also*

$$\mu \oplus^{\kappa}_{p} \mu_1 \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \nu \oplus^{\forall}_{p} \nu_1$$

Proof. This is a direct consequence of 3.7.7. □

The Guarded fundamental lemma and congruence lemma

Proof of Lemma 3.8.2. The proof proceeds by induction on $M : \mathsf{Tm}^{\Gamma}_{\sigma}$ and relies almost entirely on the compatibility lemmas with the only exception of the $M = \mathsf{lam}\, x.M'$ case.

Case: $\mathsf{lam}\, x.M'$

By induction hypothesis, we have that

$$\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma,\sigma} \delta) \to [\![M']\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(M'[\delta]).$$

Thus we get that

$$\begin{aligned}
&\forall \rho, \delta, v, V.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \wedge (v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V) \\
&\qquad \to [\![M']\!]^{\kappa}_{\rho,v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(M'[\delta, V/x]) \\
&\leftrightarrow \begin{pmatrix} \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to \forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\sigma} V) \\ \qquad \to ([\![M']\!]^{\kappa}_{\rho,v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}((M'[\delta])[V/x])) \end{pmatrix} \\
&\leftrightarrow \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![\mathsf{lam}\, x.M']\!]^{\mathsf{Val},\kappa}_{\rho} \preceq^{\kappa,\mathsf{Val}}_{\sigma\to\tau} \mathsf{lam}\, x.M'[\delta]\right) \\
&\leftrightarrow \begin{pmatrix} \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \\ \qquad \to \delta^{\kappa}([\![\mathsf{lam}\, x.M']\!]^{\mathsf{Val},\kappa}_{\rho}) \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} \delta^{\forall}(\mathsf{lam}\, x.M'[\delta]) \end{pmatrix} \\
&\leftrightarrow \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta) \to [\![\mathsf{lam}\, x.M']\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} \delta^{\forall}(\mathsf{lam}\, x.M'[\delta])\right)
\end{aligned}$$

Case: MN
Let $\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta$, by induction hypothesis we have $[\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma\to\tau} \mathsf{eval}(M[\delta])$ and $[\![N]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma} \mathsf{eval}(N[\delta])$. Now .0.10 implies that $[\![MN]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(MN[\delta])$.

Case: $\mathsf{case}(L, x.M, y.N)$
Let $\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta$, by induction hypothesis we have

1. $[\![L]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\sigma_1+\sigma_2} \mathsf{eval}(L[\delta])$
2. $\forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\sigma_1} V) \to [\![M]\!]^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(M[\delta, V/x])$
3. $\forall w, W.(w \preceq^{\kappa,\mathsf{Val}}_{\sigma_2} W) \to [\![N]\!]^{\kappa}_{\rho.w} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(N[\delta, V/y])$.

Since we can add steps to fit the requirements of .0.11, by applying this lemma we get

$$\begin{aligned}
&\left([\![L]\!]^{\kappa}_{\rho} >\!\!>\!\!=^{\kappa} \begin{cases} \mathbf{inl}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).[\![M]\!]^{\kappa}_{\rho.v}) \\ \mathbf{inr}\, w \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha:\kappa).[\![N]\!]^{\kappa}_{\rho.w}) \end{cases}\right) \\
\preceq^{\kappa,\mathsf{Tm}}_{\tau} &\left(\mathsf{eval}(L[\delta]) >\!\!>\!\!= \begin{cases} \mathsf{inl}\, V \mapsto \mathsf{step}^{\forall}(\mathsf{eval}(M[\delta][V/x])) \\ \mathsf{inr}\, V \mapsto \mathsf{step}^{\forall}(\mathsf{eval}(N[\delta][V/x])) \end{cases}\right)
\end{aligned}$$

Observe, that this is precisely

$$[\![\mathsf{case}(L, x.M, y.M)]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(\mathsf{case}(L, x.M, y.M)[\delta])$$

Case: $\mathsf{unfold}\, M$
Let $\rho \preceq^{\kappa,\mathsf{Val}}_{\Gamma} \delta$, by the induction hypothesis we have $[\![M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} \mathsf{eval}(M[\delta])$ and now .0.13 implies that $[\![\mathsf{unfold}\, M]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau\mu X.\tau/X} \mathsf{eval}(\mathsf{unfold}\, M[\delta])$

The remaining cases follow similarly. □

Proof of Lemma 3.8.4. The proof proceeds by induction on context derivations. All cases — except for the $M = \mathsf{lam}\, x.M'$ case — are direct consequences of the compatibility lemmas.

Case: $C = \mathsf{lam}\ x.C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau_1 \to \tau_2)$
Assume $M \preceq^{\kappa,\Gamma}_{\sigma} N$. We have that $C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta, (x : \tau_1) \vdash \tau_2)$ and by the induction hypothesis it follows that

$$C'[M] \preceq^{\kappa,\Gamma}_{\tau_2} C'[N]$$

It now follows that

$$\begin{aligned}
& C'[M] \preceq^{\kappa,\Gamma}_{\tau_2} C'[N] \\
\triangleq\ & \forall \rho, \delta.\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta,\tau_1} \delta \to [\![C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau_2} \mathsf{eval}((C'[N])[\delta]) \\
\leftrightarrow\ & \left(\begin{array}{l} \forall \rho, \delta, v, V. \left(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta \wedge v \preceq^{\kappa,\mathsf{Val}}_{\tau_1} V\right) \\ \qquad \to [\![C'[M]]\!]^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau_2} \mathsf{eval}((C'[N])[\delta, V/x]) \end{array}\right) \\
\leftrightarrow\ & \left(\begin{array}{l} \forall \rho, \delta (\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \\ \quad \to \left(\begin{array}{l} \forall v, V.v \preceq^{\kappa,\mathsf{Val}}_{\tau_1} V \\ \quad \to [\![C'[M]]\!]^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau_2} \mathsf{eval}((C'[N])[\delta, V/x]) \end{array}\right) \end{array}\right) \\
\leftrightarrow\ & \left(\begin{array}{l} \forall \rho, \delta (\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \\ \quad \to [\![\mathsf{lam}\ x.C'[M]]\!]^{\mathsf{Val},\kappa}_{\rho} \preceq^{\kappa,\mathsf{Val}}_{\tau_1 \to \tau_2} (\mathsf{lam}\ x.C'[N])[\delta, V/x] \end{array}\right) \\
\leftrightarrow\ & \left(\begin{array}{l} \forall \rho, \delta (\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \\ \quad \to \delta^{\kappa}([\![\mathsf{lam}\ x.C'[M]]\!]^{\mathsf{Val},\kappa}_{\rho}) \preceq^{\kappa,\mathsf{Tm}}_{\tau_1 \to \tau_2} \delta^{\forall}(\mathsf{lam}\ x.C'[N])[\delta, V/x] \end{array}\right) \\
\leftrightarrow\ & \left(\forall \rho, \delta (\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau_1 \to \tau_2} (C[N])[\delta, V/x]\right) \\
\leftrightarrow\ & C[M] \preceq^{\kappa,\Gamma}_{\tau_1 \to \tau_2} C[N]
\end{aligned}$$

Case: $C = \mathsf{fold} C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \mu X.\tau)$
Assume $M \preceq^{\kappa,\Gamma}_{\sigma} N$. We have that $C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau[\mu X.\tau/X])$ and thus by induction hypothesis we have $C'[M] \preceq^{\kappa,\Gamma}_{\tau[\mu X.\tau/X]} C'[N]$. By .0.12 we have that

$$\begin{aligned}
& C'[M] \preceq^{\kappa,\Gamma}_{\tau[\mu X.\tau/X]} C'[N] \\
\leftrightarrow\ & \left(\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau[\mu X.\tau/X]} \mathsf{eval}((C'[N])[\delta])\right) \\
\to\ & \left(\begin{array}{l} \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \\ \quad \to \left(\mathsf{D}^{\kappa}(\mathsf{next}^{\kappa})([\![C'[M]]\!]^{\kappa}_{\rho})\right) \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} \mathsf{D}^{\forall}(\mathsf{fold})\left(\mathsf{eval}\left(C'[N][\delta]\right)\right) \end{array}\right) \\
\leftrightarrow\ & \forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![\mathsf{fold} C'[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\mu X.\tau} \mathsf{eval}((\mathsf{fold} C'[N])[\delta]) \\
\leftrightarrow\ & C[M] \preceq^{\kappa,\Gamma}_{\mu X.\tau} C[N]
\end{aligned}$$

Case: $C = \mathsf{case}(L, x.C', y.N) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$
Assume that $M \preceq^{\kappa,\Gamma}_{\sigma} M'$. We furthermore have that

1. $\Delta \vdash L : \tau_1 + \tau_2$
2. $C' : (\Gamma \vdash \sigma) \Rightarrow (\Delta, x : \tau_1 \vdash \tau)$

3. $\Delta, y : \tau_2 \vdash N : \tau$

By the induction hypothesis and 3.8.2 it follows directly that $C'[M] \preceq^{\kappa,\Gamma}_{\tau} C'[M']$, $L \preceq^{\kappa,\Gamma}_{\tau_1+\tau_2} L$ and $N \preceq^{\kappa,\Gamma}_{\tau} N$. Thus, for all ρ and δ such that $\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta$, we get that

$$
\begin{aligned}
&(1) \quad [\![L]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau_1+\tau_2} \mathsf{eval}(L[\delta]) \\
&(2) \quad \forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\tau_1} V) \\
&\qquad\qquad \to [\![C'[M]]\!]^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}((C'[M'])[\delta, V/x]) \\
&(3) \quad \forall v, V.(v \preceq^{\kappa,\mathsf{Val}}_{\tau_2} V) \to [\![N]\!]^{\kappa}_{\rho.v} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(N[\delta, V/x])
\end{aligned}
$$

Now, using .0.11 we conclude that

$$
\begin{aligned}
&[\![L]\!]^{\kappa}_{\rho} >\!>\!=^{\kappa} \begin{cases} \mathbf{inl}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha : \kappa).[\![C'[M]]\!]^{\kappa}_{\rho.v}) \\ \mathbf{inr}\, v \mapsto \mathsf{step}^{\kappa}(\lambda(\alpha : \kappa).[\![N]\!]^{\kappa}_{\rho.v}) \end{cases} \\
&\preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(\mathsf{case}(L, x.C'[M'], y.N)[\delta])
\end{aligned}
$$

and thus we get that

$$\forall \rho, \delta.(\rho \preceq^{\kappa,\mathsf{Val}}_{\Delta} \delta) \to [\![C[M]]\!]^{\kappa}_{\rho} \preceq^{\kappa,\mathsf{Tm}}_{\tau} \mathsf{eval}(C[M'])$$

Case: $C = \mathsf{choice}^{p}(C', N) : (\Gamma \vdash \sigma) \Rightarrow (\Delta \vdash \tau)$
Assume $M \preceq^{\kappa,\Gamma}_{\sigma} M'$. It follows from the assumptions that furthermore $\Delta \vdash N : \sigma$ and thus by 3.8.2 we get $N \preceq^{\kappa,\Gamma}_{\sigma} N$. Now the induction hypothesis implies $C'[M] \preceq^{\kappa,\Delta}_{\sigma} C'[M']$ and consequently $C[M] \preceq^{\kappa,\Delta}_{\tau} C[M']$ follows by .0.14.

The remaining cases are similar. □

Section 3.9

Proof of Lemma 3.9.2. Using (3.7) we compute

$$
\begin{aligned}
[\![\mathsf{hid}_{\mathsf{p}}]\!]^{\mathsf{Val},\kappa}(v) &= [\![\mathsf{Y}(\mathsf{hid}'_{\mathsf{p}})\, z]\!]^{\kappa}_{z \mapsto v} \\
&= (\Delta^{\kappa})^4 [\![\mathsf{hid}'_{\mathsf{p}}(\mathsf{Y}(\mathsf{hid}'_{\mathsf{p}}))\, z]\!]^{\kappa}_{z \mapsto v}
\end{aligned}
$$

Note that by definition of Y there exists a value $W_{\mathsf{hid},\mathsf{p}}$ such that

$$[\![\mathsf{Y}(\mathsf{hid}'_{\mathsf{p}})]\!]^{\kappa} = \mathsf{step}^{\kappa}(\mathsf{next}^{\kappa}([\![W_{\mathsf{hid},\mathsf{p}}]\!]^{\kappa}))$$

and so

$$
\begin{aligned}
[\![\mathsf{hid}_{\mathsf{p}}]\!]^{\kappa}(v) &= (\Delta^{\kappa})^7 [\![\mathsf{choice}^{p}(x, fx)]\!]^{\kappa}_{x \mapsto v, f \mapsto [\![W_{\mathsf{hid},\mathsf{p}}]\!]^{\mathsf{Val},\kappa}} \\
&= (\Delta^{\kappa})^7 (v \oplus^{\kappa}_{p} [\![W_{\mathsf{hid},\mathsf{p}} x]\!]^{\kappa}_{x \mapsto v})
\end{aligned}
$$

Therefore $[\![\mathsf{hid}_{\mathsf{p}}]\!]^{\mathsf{Val},\kappa}(v) \preceq^{\kappa,\mathsf{Val}}_{\sigma} \mu$ unfolds to

$$(\triangleright^{\kappa})^5 \left((\Delta^{\kappa})^2 (v \oplus^{\kappa}_{p} [\![W_{\mathsf{hid},\mathsf{p}} x]\!]^{\kappa}_{x \mapsto v}) \preceq^{\kappa,\mathsf{Val}}_{\sigma} \mu \right)$$

which by Lemma 3.7.6 is equivalent to

$$(\triangleright^\kappa)^5\left(((\Delta^\kappa)^2 v) \oplus_p^\kappa (\Delta^\kappa)^2([\![W_{\mathsf{hid},\mathsf{p}} x]\!]^\kappa_{x\mapsto v})) \preceq_\sigma^{\kappa,\mathsf{Val}} \mu\right)$$

Since $(\Delta^\kappa)^2([\![W_{\mathsf{hid},\mathsf{p}} x]\!]^\kappa_{x\mapsto v}) = [\![\mathsf{hid}_\mathsf{p}]\!]^{\mathsf{Val},\kappa}(v)$, this concludes the proof. □

Section 3.9

Proof of Theorem 3.9.3. As in Section 3.9, note that there is a value $W_{\mathsf{efair},\mathsf{p}}$ such that $\mathsf{eval}^\kappa(\mathsf{Y}(\mathsf{efair}'_\mathsf{p})) = \Delta^\kappa(\mathsf{eval}^\kappa(W_{\mathsf{efair},\mathsf{p}}))$. Then

$$\begin{aligned}
&\mathsf{eval}^\kappa(\mathsf{efair}_\mathsf{p}\langle\rangle)\\
=&(\Delta^\kappa)^4\left(\mathsf{eval}(\mathsf{efair}'_\mathsf{p}(\mathsf{Y}(\mathsf{efair}'_\mathsf{p}))\langle\rangle)\right)\\
=&(\Delta^\kappa)^6\begin{pmatrix}(\mathsf{tt}\oplus_p^\kappa \mathsf{ff}) >\!\!>\!\!=^\kappa \lambda v.\Delta^\kappa(\mathsf{tt}\oplus_p^\kappa \mathsf{ff}) >\!\!>\!\!=^\kappa\\ \lambda w.\Delta^\kappa\mathsf{eval}(\mathsf{if}\ \mathsf{eqbool}(v,w)\ \mathsf{then}\ W_{\mathsf{efair},\mathsf{p}}\langle\rangle\ \mathsf{else}\ v)\end{pmatrix}\\
=&(\Delta^\kappa)^6\begin{pmatrix}\Delta^\kappa\left((\Delta^\kappa(\mathsf{eval}(W_{\mathsf{efair},\mathsf{p}}\langle\rangle)))\oplus_p^\kappa(\Delta^\kappa\mathsf{tt})\right)\\ \oplus_p\\ \Delta^\kappa\left((\Delta^\kappa\mathsf{ff})\oplus_p^\kappa\Delta^\kappa(\Delta^\kappa(\mathsf{eval}(W_{\mathsf{efair},\mathsf{p}}\langle\rangle)))\right)\end{pmatrix}\\
=&(\Delta^\kappa)^6\begin{pmatrix}\Delta^\kappa\left(\mathsf{eval}(\mathsf{Y}(\mathsf{efair}'_\mathsf{p})\langle\rangle)\oplus_p^\kappa(\Delta^\kappa\mathsf{tt})\right)\\ \oplus_p\\ \Delta^\kappa\left((\Delta^\kappa\mathsf{ff})\oplus_p^\kappa\mathsf{eval}(\mathsf{Y}(\mathsf{efair}'_\mathsf{p})\langle\rangle)\right)\end{pmatrix}\\
=&(\Delta^\kappa)^6\begin{pmatrix}\Delta^\kappa\left(\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)\oplus_p^\kappa(\Delta^\kappa\mathsf{tt})\right)\\ \oplus_p\\ \Delta^\kappa\left((\Delta^\kappa\mathsf{ff})\oplus_p^\kappa\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)\right)\end{pmatrix}
\end{aligned}$$

So that

$$\begin{aligned}
&\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)\\
\rightsquigarrow&(\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)\oplus_p^\forall \mathsf{tt})\oplus_p^\forall(\mathsf{ff}\oplus_p^\forall\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle))\\
=&(\mathsf{tt}\oplus^\forall_{2p(1-p)}(\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)))\oplus^\forall_{\frac{1}{2}}(\mathsf{ff}\oplus^\forall_{2p(1-p)}(\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)))
\end{aligned}$$

On the other hand,

$$[\![\mathsf{hfair}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^\kappa = \Delta^\kappa([\![\mathsf{hid}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^\kappa(\mathsf{tt})\oplus^\kappa_{\frac{1}{2}}[\![\mathsf{hid}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^\kappa(\mathsf{ff}))$$

We will show that $[\![\mathsf{hfair}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)$ by guarded recursion. So assume $\triangleright^\kappa\left([\![\mathsf{hfair}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)\right)$. By the above and Lemmas 3.7.5 and 3.7.7 it suffices to show

$$\triangleright^\kappa\left([\![\mathsf{hid}_{2\cdot\mathsf{p}(1-\mathsf{p})}]\!]^\kappa(\mathsf{tt}) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} (\mathsf{tt}\oplus^\forall_{2p(1-p)}(\mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle)))\right)$$

and similarly for ff. By Lemma 3.9.2, using Lemma 3.7.7, this reduces to showing $\triangleright^\kappa(\mathsf{tt} \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{tt})$ and $\triangleright^\kappa([\![\mathsf{hid}_{2\cdot \mathsf{p}(1-\mathsf{p})}]\!]^\kappa(\mathsf{tt}) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{efair}_\mathsf{p}\langle\rangle))$. The former of this follows from Lemma 3.8.2 and the latter is the guarded recursion hypothesis.

For the other direction, we again reason by guarded recursion. Applying the soundness theorem (Theorem 3.6.1) to the above gives

$$[\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa = (\Delta^\kappa)^6 \begin{pmatrix} \Delta^\kappa\left([\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa \oplus^\kappa_p (\Delta^\kappa \mathsf{tt})\right) \\ \oplus_p \\ \Delta^\kappa\left((\Delta^\kappa \mathsf{ff}) \oplus^\kappa_p [\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa\right) \end{pmatrix}$$

and therefore $[\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{hid}_{2\cdot \mathsf{p}(1-\mathsf{p})})$ is equivalent to $(\triangleright^\kappa)^7$ applied to

$$\begin{pmatrix} \left([\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa \oplus^\kappa_p (\Delta^\kappa \mathsf{tt})\right) \\ \oplus_p \\ \left((\Delta^\kappa \mathsf{ff}) \oplus^\kappa_p [\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa\right) \end{pmatrix} \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{hid}_{2\cdot \mathsf{p}(1-\mathsf{p})})$$

Note that

$$\begin{aligned} &\left([\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa \oplus^\kappa_p (\Delta^\kappa \mathsf{tt})\right) \oplus^\kappa_p \left((\Delta^\kappa \mathsf{ff}) \oplus^\kappa_p [\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa\right) \\ &= \left((\Delta^\kappa \mathsf{tt}) \oplus^\kappa_{2p(1-p)} [\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa\right) \oplus^\kappa_{\frac{1}{2}} \left((\Delta^\kappa \mathsf{ff}) \oplus^\kappa_{2p(1-p)} [\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa\right) \end{aligned}$$

A calculation similar to the proof of Lemma 3.9.2 shows that

$$\mathsf{eval}(\mathsf{hid}_{2\mathsf{p}(1-\mathsf{p})})$$

can be related in the symmetric, transitive closure of $\rightsquigarrow$ to

$$(\mathsf{tt} \oplus^\forall_{2p(1-p)} \mathsf{eval}(\mathsf{hid}_{2\mathsf{p}(1-\mathsf{p})})) \oplus^\forall_{\frac{1}{2}} (\mathsf{ff} \oplus^\forall_{2p(1-p)} \mathsf{eval}(\mathsf{hid}_{2\mathsf{p}(1-\mathsf{p})}))$$

Putting this together, the goal $[\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} \mathsf{eval}(\mathsf{hid}_{2\cdot \mathsf{p}(1-\mathsf{p})})$ reduces to proving

$$(\triangleright^\kappa)^7\left(\left((\Delta^\kappa \mathsf{tt}) \oplus^\kappa_q [\![\mathsf{efair}_\mathsf{p}\langle\rangle]\!]^\kappa\right) \preceq^{\kappa,\mathsf{Tm}}_{\mathsf{bool}} (\mathsf{tt} \oplus^\kappa_q \mathsf{eval}(\mathsf{hid}_\mathsf{q}))\right)$$

where $q = 2p(1-p)$ and similarly for ff. This follows easily from the guarded recursion hypothesis. □

Section 3.9

The proof of Theorem 3.9.4 uses Lemma .0.16 and Lemma .0.17 below.

Lemma .0.15 (Geometric Sum)**.** *Let $a : \mathbb{Q}$ and $r : (0,1)$ be arbitrary, then*

$$\Sigma_{k=0}^{n-1} a \cdot r^k = a(\tfrac{1-r^n}{1-r})$$

In particular, if $a = p$ and $r = (1-p)$ with $p : (0,1)$, we get that

$$\Sigma_{k=0}^{n-1} p \cdot (1-p)^k = p(\tfrac{1-(1-p)^n}{p}) = 1-(1-p)^n$$

Lemma .0.16. *(Geometric series convergence) For any rational $\varepsilon : (0,1)$ exists an $n : \mathscr{N}$ such that $1 - \sum_{k=0}^{n-1} p(1-p)^k < \varepsilon$.*

Proof. Without loss of generality we may assume that $\varepsilon = \frac{1}{2^k}$. Indeed, we can prove that

$$\forall n, m. \frac{n}{m} : (0,1). \exists k. \frac{1}{2^k} < \frac{n}{m}.$$

Since $\frac{1}{m} \leq \frac{n}{m}$ it suffices to show that $\forall m. \exists k. 2^k > m$ (which follows by induction on m).

Our goal is to prove that $\forall k. \exists n. 1 - \sum_{i=0}^{n-1} p(1-p)^i < \frac{1}{2^k}$. Considering that $1 - \sum_{i=0}^{n-1} p(1-p)^i = (1-p)^n$ and $(1-p) = \frac{p_N}{p_D}$ for some integers p_N and p_D with $p_N < p_D$ we get that

$$1 - \sum_{i=0}^{n-1} p(1-p)^i < \frac{1}{2^k} \quad \leftrightarrow \quad \left(\frac{p_N}{p_D}\right)^n < \frac{1}{2^k} \quad \leftrightarrow \quad 2^k < \left(\frac{p_D}{p_N}\right)^n$$

But since $p_D \geq p_N + 1$ and furthermore for all integers we have that $\left(\frac{n+1}{n}\right)^n \geq 2$, we get

$$\left(\frac{p_D}{p_N}\right)^{p_N \cdot k} \geq \left(\left(\frac{p_N + 1}{p_N}\right)^{p_N}\right)^k \geq 2^k$$

Thus, for any p_N, p_D with $(1-p) = \frac{p_N}{p_D}$ and $k : \mathscr{N}$ the number $n \triangleq p_N \cdot k + 1$ witnesses

$$\forall k. \exists n. 1 - \sum_{i=0}^{n-1} p(1-p)^i < \frac{1}{2^k}$$

□

Lemma .0.17. *For any value V, the element $\mathsf{eval}(\mathsf{hid}_\mathsf{p} V)$ is related in the symmetric transitive closure of $\rightsquigarrow$ to $\delta^\forall(V) \oplus_p^\forall \mathsf{eval}(\mathsf{hid}_\mathsf{p} V)$.*

Proof. Let $W_{\mathsf{hid},\mathsf{p}}$ be the value such that

$$\mathsf{eval}(\mathsf{Y}(\mathsf{hid}'_\mathsf{p})) = \mathsf{step}^\forall(\delta^\kappa(W_{\mathsf{hid},\mathsf{p}}))$$

Then

$$\begin{aligned}
\mathsf{eval}(\mathsf{hid}_\mathsf{p} V) &= \mathsf{eval}(\mathsf{Y}(\mathsf{hid}'_\mathsf{p}) V) \\
&\rightsquigarrow \mathsf{eval}(\mathsf{hid}'_\mathsf{p}(\mathsf{Y}(\mathsf{hid}'_\mathsf{p})) V) \\
&\rightsquigarrow \mathsf{eval}(\mathsf{hid}'_\mathsf{p} W_{\mathsf{hid},\mathsf{p}} V) \\
&\rightsquigarrow \mathsf{eval}(\mathsf{choice}^p(V, W_{\mathsf{hid},\mathsf{p}} V)) \\
&= \delta^\forall(V) \oplus_p^\forall \mathsf{eval}(W_{\mathsf{hid},\mathsf{p}} V)
\end{aligned}$$

and since

$$\begin{aligned}
\mathsf{eval}(\mathsf{hid}_\mathsf{p} V) &= \mathsf{eval}(\mathsf{Y}(\mathsf{hid}'_\mathsf{p}) V) \\
&\rightsquigarrow \mathsf{eval}(W_{\mathsf{hid},\mathsf{p}} V)
\end{aligned}$$

the result follows. □

Bibliography

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 147–160, New York, NY, USA, January 1999. Association for Computing Machinery. ISBN 978-1-58113-095-9. doi: 10.1145/292540.292555. URL https://dl.acm.org/doi/10.1145/292540.292555.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1): 3–27, September 2005. ISSN 0304-3975. doi: 10.1016/j.tcs.2005.06.002. URL https://www.sciencedirect.com/science/article/pii/S0304397505003373.

[3] Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity*. Habilitation, 2013.

[4] Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. *ACM SIGPLAN Notices*, 48(1):27–38, January 2013. ISSN 0362-1340. doi: 10.1145/2480359.2429075. URL https://doi.org/10.1145/2480359.2429075.

[5] Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL), dec 2017. doi: 10.1145/3158111.

[6] Andreas M. Abel and Brigitte Pientka. Wellfounded recursion with copatterns: a unified approach to termination and productivity. *ACM SIGPLAN Notices*, 48(9):185–196, September 2013. ISSN 0362-1340. doi: 10.1145/2544174.2500591. URL https://doi.org/10.1145/2544174.2500591.

[7] Alejandro Aguirre and Lars Birkedal. Step-indexed logical relations for countable nondeterminism and probabilistic choice. *Proc. ACM Program. Lang.*, 7 (POPL):33–60, 2023. doi: 10.1145/3571195. URL https://doi.org/10.1145/3571195.

[8] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor Mcbride, and Peter Morris. Indexed containers. *Journal of Functional Programming*, 25:e5, January 2015. ISSN 0956-7968, 1469-7653. doi: 10.1017/S095679681500009X. URL https://www.cambridge.org/core/journals/journal-of-functional-programming/article/indexed-containers/FB9C7DC88A65E7529D39554379D9765F.

[9] Thorsten Altenkirch, Nils Anders Danielsson, and Nicolai Kraus. Partiality, revisited: The partiality monad as a quotient inductive-inductive type. In Javier Esparza and Andrzej S. Murawski, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 534–549. Springer, March 2017. ISBN 978-3-662-54457-0. doi: 10.1007/978-3-662-54458-7_31.

[10] Pierre America and Jan J. M. M. Rutten. Solving reflexive domain equations in a category of complete metric spaces. *J. Comput. Syst. Sci.*, 39(3):343–375, 1989. doi: 10.1016/0022-0000(89)90027-5. URL https://doi.org/10.1016/0022-0000(89)90027-5.

[11] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Trans. Program. Lang. Syst.*, 23(5):657–683, 2001. doi: 10.1145/504709.504712. URL https://doi.org/10.1145/504709.504712.

[12] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 109–122. ACM, 2007. doi: 10.1145/1190216.1190235. URL https://doi.org/10.1145/1190216.1190235.

[13] Robert Atkey and Conor McBride. Productive Coprogramming with Guarded Recursion. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 197–208. Association for Computing Machinery, 2013. doi: 10.1145/2500365.2500597. URL https://doi.org/10.1145/2500365.2500597.

[14] Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017. doi: 10.1109/LICS.2017.8005097. URL http://www.itu.dk/people/mogel/papers/lics2017.pdf.

[15] Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Simply ratt: A fitch-style modal calculus for reactive programming without space leaks.

Proc. ACM Program. Lang., 3:109:1–109:27, 2019. ISSN 2475-1421. doi: 10.1145/3341713.

[16] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 90–101. ACM, 2009. doi: 10.1145/1480881.1480894. URL https://doi.org/10.1145/1480881.1480894.

[17] Gilles Barthe, Thomas Espitau, Benjamin Grégoire, Justin Hsu, Léo Stefanesco, and Pierre-Yves Strub. Relational reasoning via probabilistic coupling. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 387–401. Springer, 2015. doi: 10.1007/978-3-662-48899-7_27. URL https://doi.org/10.1007/978-3-662-48899-7_27.

[18] Nick Benton, Andrew Kennedy, and Carsten Varming. Some Domain Theory and Denotational Semantics in Coq. 5674:115–130, August 2009. doi: 10.1007/978-3-642-03359-9_10.

[19] Nick Benton, Lars Birkedal, Andrew Kennedy, and Carsten Varming. Formalizing Domains, Ultrametric Spaces and Semantics of Programming Languages. July 2010. URL https://www.microsoft.com/en-us/research/publication/formalizing-domains-ultrametric-spaces-and-semantics-of-programming-languages/.

[20] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda -calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, July 1991. doi: 10.1109/LICS.1991.151645. URL https://ieeexplore.ieee.org/document/151645.

[21] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 213–222. IEEE Computer Society, 2013. doi: 10.1109/LICS.2013.27. URL https://doi.org/10.1109/LICS.2013.27.

[22] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional Type Theory with Guarded Recursive Types qua Fixed Points on Universes. In *2013 28th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 213–222, June 2013. doi: 10.1109/LICS.2013.27. URL https://ieeexplore.ieee.org/document/6571553. ISSN: 1043-6871.

[23] Lars Birkedal, Rasmus Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012. doi: 10.2168/LMCS-8(4:1)2012.

[24] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded cubical type theory. *Journal of Automated Reasoning*, (63):211–253, 2019.

[25] Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science*, 30(2):118–138, 2020. doi: 10.1017/S0960129519000197.

[26] Ales Bizjak and Lars Birkedal. Step-indexed logical relations for probability. In Andrew M. Pitts, editor, *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9034 of *Lecture Notes in Computer Science*, pages 279–294. Springer, 2015. doi: 10.1007/978-3-662-46678-0_18. URL https://doi.org/10.1007/978-3-662-46678-0_18.

[27] Aleš Bizjak and Rasmus Ejlers Møgelberg. A model of guarded recursion with clock synchronisation. *Electronic Notes in Theoretical Computer Science*, 319:83 – 101, 2015. doi: https://doi.org/10.1016/j.entcs.2015.12.007. URL http://www.sciencedirect.com/science/article/pii/S1571066115000742. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI).

[28] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus E. Møgelberg, and Lars Birkedal. Guarded Dependent Type Theory with Coinductive Types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 20–35. Springer Berlin Heidelberg, 2016.

[29] Venanzio Capretta. General Recursion via Coinductive Types. *Logical Methods in Computer Science*, Volume 1, Issue 2, July 2005. ISSN 1860-5974. doi: 10.2168/LMCS-1(2:1)2005. URL https://lmcs.episciences.org/2265. Publisher: Episciences.org.

[30] Venanzio Capretta, Tarmo Uustalu, and Varmo Vene. Corecursive Algebras: A Study of General Structured Corecursion. In Marcel Vinícius Medeiros Oliveira and Jim Woodcock, editors, *Formal Methods: Foundations and Applications*, Lecture Notes in Computer Science, pages 84–100, Berlin, Heidelberg, 2009. Springer. ISBN 978-3-642-10452-7. doi: 10.1007/978-3-642-10452-7_7.

[31] Luca Castiglione, Philipp Stassen, Cora Perner, Daniel Pereira, Gustavo Bertoli, and Emil Lupu. Don't Panic! Analysing the Impact of Attacks on the Safety of Flight Management Systems. pages 1–10, October 2023. doi: 10.1109/DASC58513.2023.10311328.

[32] Evan Cavallo and Robert Harper. Higher inductive types in cubical computational type theory. *Proceedings of the ACM on Programming Languages*, 3(POPL), January 2019. doi: https://doi.org/10.1145/3290314. URL https://dl.acm.org/doi/10.1145/3290314.

[33] Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. Sikkel: Multimode simple type theory as an agda library. In *Electronic Proceedings in Theoretical Computer Science*, volume 360, pages 93–112. Munich, Germany, Open Publishing Association, 06 2022. doi: 10.4204/EPTCS.360.5.

[34] James Chapman, Tarmo Uustalu, and Niccolò Veltri. Quotienting the Delay Monad by Weak Bisimilarity. In Martin Leucker, Camilo Rueda, and Frank D. Valencia, editors, *Theoretical Aspects of Computing - ICTAC 2015*, Lecture Notes in Computer Science, pages 110–125, Cham, 2015. Springer International Publishing. ISBN 978-3-319-25150-9. doi: 10.1007/978-3-319-25150-9_8.

[35] Nicolas Chappe, Paul He, Ludovic Henrio, Yannick Zakowski, and Steve Zdancewic. Choice trees: Representing nondeterministic, recursive, and impure programs in coq. *Proc. ACM Program. Lang.*, 7(POPL):1770–1800, 2023. doi: 10.1145/3571254. URL https://doi.org/10.1145/3571254.

[36] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 407–421. Springer Berlin Heidelberg, 2015. ISBN 978-3-662-46678-0.

[37] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. *FLAP*, 4(10): 3127–3170, 2017. URL http://collegepublications.co.uk/ifcolog/?00019.

[38] T. Coquand. Pattern Matching with Dependent Types. 1992. URL https://www.semanticscholar.org/paper/Pattern-Matching-with-Dependent-Types-Coquand/93b3b5832d8bc558c31a54dff5f083cb2f9c04d1.

[39] Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, Lecture Notes in Computer Science, pages 62–78, Berlin, Heidelberg, 1994. Springer. ISBN 978-3-540-48440-0. doi: 10.1007/3-540-58085-9_72.

[40] Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996. ISSN 0167-6423. doi: https://doi.org/10.1016/0167-6423(95)00021-6.

[41] Raphaëlle Crubillé and Ugo Dal Lago. On probabilistic applicative bisimulation and call-by-value λ-calculi. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2014. doi: 10.1007/978-3-642-54833-8_12. URL https://doi.org/10.1007/978-3-642-54833-8_12.

[42] Ryan Culpepper and Andrew Cobb. Contextual equivalence for probabilistic programs with continuous random variables and scoring. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 368–392. Springer, 2017. doi: 10.1007/978-3-662-54434-1_14. URL https://doi.org/10.1007/978-3-662-54434-1_14.

[43] Fredrik Dahlqvist and Dexter Kozen. Semantics of higher-order probabilistic programs with conditioning. *Proc. ACM Program. Lang.*, 4(POPL):57:1–57:29, 2020. doi: 10.1145/3371125. URL https://doi.org/10.1145/3371125.

[44] Rowan Davies and Frank Pfenning. A modal analysis of staged computation. *Journal of the ACM*, 48(3):555–604, May 2001.

[45] Tom de Jong and Martín Hötzel Escardó. Domain Theory in Constructive and Predicative Univalent Foundations. In *DROPS-IDN/v2/document/10.4230/LIPIcs.CSL.2021.28*. Schloss-Dagstuhl - Leibniz Zentrum für Informatik, 2021. doi: 10.4230/LIPIcs.CSL.2021.28. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2021.28.

[46] Robert Dockins. Formalized, Effective Domain Theory in Coq. URL https://www.springerprofessional.de/en/formalized-effective-domain-theory-in-coq/2199494.

[47] Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf's type theory. *Theoretical Computer Science*, 176(1):329–335, April 1997. ISSN 0304-3975. doi: 10.1016/S0304-3975(96)00145-4. URL https://www.sciencedirect.com/science/article/pii/S0304397596001454.

[48] Thomas Ehrhard, Michele Pagani, and Christine Tasson. Full abstraction for probabilistic PCF. *J. ACM*, 65(4):23:1–23:44, 2018. doi: 10.1145/3164540. URL https://doi.org/10.1145/3164540.

[49] M.H. Escardo. A metric model of pcf. Presented at the Workshop on Realizability Semantics and Applications, June 30-July 1, 1999., 1999. URL https://www.cs.bham.ac.uk/~mhe/papers/metricpcf.pdf.

[50] Martín Hötzel Escardó. A metric model of PCF. 1998. URL https://api.semanticscholar.org/CorpusID:13035272.

[51] Marcelo P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge University Press, 1996.

[52] Dan Frumin, Amin Timany, and Lars Birkedal. Modular denotational semantics for effects with guarded interaction trees. *Proc. ACM Program. Lang.*, (POPL), 2024.

[53] Eduarde Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, Lecture Notes in Computer Science, pages 39–59, Berlin, Heidelberg, 1995. Springer. ISBN 978-3-540-47770-9. doi: 10.1007/3-540-60579-7_3.

[54] Daniel Gratzer. Normalization for multimodal type theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393515. doi: 10.1145/3531130.3532398. URL https://doi.org/10.1145/3531130.3532398.

[55] Daniel Gratzer. *Syntax and semantics of modal type theory*. PhD thesis, Aarhus University, August 2023.

[56] Daniel Gratzer and Lars Birkedal. A Stratified Approach to Löb Induction. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:22, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISBN 978-3-95977-233-4. doi: 10.4230/LIPIcs.FSCD.2022.23. URL https://drops.dagstuhl.de/opus/volltexte/2022/16304.

[57] Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a Modal Dependent Type Theory. *Proc. ACM Program. Lang.*, 3, 2019. doi: 10.1145/3341711.

[58] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume

17, Issue 3, July 2021. doi: 10.46298/lmcs-17(3:11)2021. URL https://lmcs.episciences.org/7713.

[59] Simon Oddershede Gregersen, Alejandro Aguirre, Philipp G. Haselwarter, Joseph Tassarotti, and Lars Birkedal. Asynchronous probabilistic couplings in higher-order separation logic. *Proc. ACM Program. Lang.*, 8(POPL), jan 2024. doi: 10.1145/3632868. URL https://doi.org/10.1145/3632868.

[60] Adrien Guatto. A generalized modality for recursion. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18. ACM, 2018. doi: 10.1145/3209108.3209148.

[61] Chris Heunen, Ohad Kammar, Sam Staton, and Hongseok Yang. A convenient category for higher-order probability theory. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–12. IEEE Computer Society, 2017. doi: 10.1109/LICS.2017.8005137. URL https://doi.org/10.1109/LICS.2017.8005137.

[62] Martin Hofmann. Syntax and Semantics of Dependent Types. In Andrew M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997. doi: 10.1017/CBO9780511526619.004. URL https://www.tcs.ifi.lmu.de/mitarbeiter/martin-hofmann/pdfs/syntaxandsemanticsof-dependenttypes.pdf.

[63] Jason Z. S. Hu and Brigitte Pientka. An investigation of kripke-style modal type theories, 2022. URL https://arxiv.org/abs/2206.07823.

[64] Martin Hyland. First steps in synthetic domain theory. pages 131–156, 1991. doi: 10.1007/BFb0084217.

[65] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of The European Association for Theoretical Computer Science*, 1997. URL https://www.semanticscholar.org/paper/A-tutorial-on-(co)algebras-and-(co)induction-Jacobs-Rutten/40bbe9978e2c4080740f55634ac58033bfb37d36.

[66] Bart Jacobs. Convexity, duality and effects. In Cristian S. Calude and Vladimiro Sassone, editors, *Theoretical Computer Science - 6th IFIP TC 1/WG 2.2 International Conference, TCS 2010, Held as Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings*, volume 323 of *IFIP Advances in Information and Communication Technology*, pages 1–19. Springer, 2010. doi: 10.1007/978-3-642-15240-5_1. URL https://doi.org/10.1007/978-3-642-15240-5_1.

[67] Patricia Johann, Alex Simpson, and Janis Voigtländer. A generic operational metatheory for algebraic effects. In *Proceedings of the 25th Annual IEEE*

Symposium on Logic in Computer Science, LICS 2010, 11-14 July 2010, Edinburgh, United Kingdom, pages 209–218. IEEE Computer Society, 2010. doi: 10.1109/LICS.2010.29. URL https://doi.org/10.1109/LICS.2010.29.

[68] C. Jones and Gordon D. Plotkin. A probabilistic powerdomain of evaluations. In *Proceedings of the Fourth Annual Symposium on Logic in Computer Science (LICS '89), Pacific Grove, California, USA, June 5-8, 1989*, pages 186–195. IEEE Computer Society, 1989. doi: 10.1109/LICS.1989.39173. URL https://doi.org/10.1109/LICS.1989.39173.

[69] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 637–650, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676980. URL https://doi.org/10.1145/2676726.2676980. event-place: Mumbai, India.

[70] G. A. Kavvos. Modalities, cohesion, and information flow. *Proceedings of the ACM on Programming Languages*, 3:20:1–20:29, 2019. doi: 10.1145/3290333.

[71] Neelakantan R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In *Proceedings of the 18th ACM SIGPLAN international conference on Functional programming*, ICFP '13, pages 221–232, New York, NY, USA, September 2013. Association for Computing Machinery. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500588. URL https://doi.org/10.1145/2500365.2500588.

[72] Magnus Kristensen, Rasmus Ejlers Mogelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450393515. doi: 10.1145/3531130.3533359. URL https://doi.org/10.1145/3531130.3533359.

[73] Ugo Dal Lago, Davide Sangiorgi, and Michele Alberti. On coinductive equivalences for higher-order probabilistic functional programs. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 297–308. ACM, 2014. doi: 10.1145/2535838.2535872. URL https://doi.org/10.1145/2535838.2535872.

[74] F. William Lawvere. Axiomatic cohesion. *Theory and Applications of Categories*, 19(3):41–49, 2007. URL http://www.tac.mta.ca/tac/volumes/19/3/19-03.pdf.

[75] Daniel R. Licata and Michael Shulman. Adjoint Logic with a 2-Category of Modes. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, pages 219–235. Springer International Publishing, 2016. doi: 10.1007/978-3-319-27683-0_16.

[76] Daniel R. Licata, Michael Shulman, and Mitchell Riley. A Fibrational Framework for Substructural and Modal Logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:22. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. doi: 10.4230/LIPIcs.FSCD.2017.25.

[77] Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal Universes in Models of Homotopy Type Theory. In H. Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 22:1–22:17. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. doi: 10.4230/LIPIcs.FSCD.2018.22.

[78] T. Lindvall. *Lectures on the Coupling Method*. Dover Books on Mathematics Series. Dover Publications, Incorporated, 2002. ISBN 978-0-486-42145-2.

[79] Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):159–208, July 2020. ISSN 0305-0041, 1469-8064. doi: 10.1017/S030500411900015X. URL http://arxiv.org/abs/1705.07088. arXiv:1705.07088 [math].

[80] Bassel Mannaa, Rasmus Ejlers Møgelberg, and Niccolò Veltri. Ticking clocks as dependent right adjoints: Denotational semantics for clocked type theory. *Logical Methods in Computer Science*, Volume 16, Issue 4, December 2020. ISSN 1860-5974. doi: 10.23638/LMCS-16(4:17)2020. URL https://lmcs.episciences.org/6980. Publisher: Episciences.org.

[81] Rasmus Ejlers Møgelberg. A type theory for productive coprogramming via guarded recursion. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, CSL-LICS '14, 2014. doi: 10.1145/2603088.2603132.

[82] Rasmus Ejlers Møgelberg and Marco Paviotti. Denotational semantics of recursive types in synthetic guarded domain theory. *Math. Struct. Comput. Sci.*, 29(3):465–510, 2019. doi: 10.1017/S0960129518000087. URL https://doi.org/10.1017/S0960129518000087.

[83] Rasmus Ejlers Møgelberg and Niccolò Veltri. Bisimulation as path type for guarded recursive types. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019. doi: 10.1145/3290317. URL https://doi.org/10.1145/3290317.

[84] Rasmus Ejlers Møgelberg and Andrea Vezzosi. Two guarded recursive powerdomains for applicative simulation. In Ana Sokolova, editor, *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics, MFPS 2021, Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021*, volume 351 of *EPTCS*, pages 200–217, 2021. doi: 10.4204/EPTCS.351.13. URL https://doi.org/10.4204/EPTCS.351.13.

[85] Rasmus Ejlers Møgelberg and Maaike Zwart. What monads can and cannot do with a bit of extra time. *CoRR*, abs/2311.15919, 2023. doi: 10.48550/ARXIV.2311.15919. URL https://doi.org/10.48550/arXiv.2311.15919.

[86] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991. ISSN 0890-5401. doi: 10.1016/0890-5401(91)90052-4. Selections from 1989 IEEE Symposium on Logic in Computer Science.

[87] Tom Murphy, VII. *Modal Types for Mobile Code*. PhD Thesis, Carnegie Mellon, January 2008. URL http://tom7.org/papers/.

[88] Rasmus Ejlers Møgelberg and Andrea Vezzosi. Two Guarded Recursive Powerdomains for Applicative Simulation. In Ana Sokolova, editor, \rm *Proceedings 37th Conference on Mathematical Foundations of Programming Semantics,* \rm *Hybrid: Salzburg, Austria and Online, 30th August - 2nd September, 2021*, volume 351 of *Electronic Proceedings in Theoretical Computer Science*, pages 200–217. Open Publishing Association, 2021. doi: 10.4204/EPTCS.351.13.

[89] H. Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266. IEEE Computer Society, 2000.

[90] Andreas Nuyts. Menkar. https://github.com/anuyts/menkar, 2019.

[91] Marco Paviotti, Rasmus Ejlers Møgelberg, and Lars Birkedal. A model of PCF in guarded type theory. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 333–349. Elsevier, 2015. doi: 10.1016/J.ENTCS.2015.12.020. URL https://doi.org/10.1016/j.entcs.2015.12.020.

[92] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 221–230. IEEE, 2001. doi: 10.1109/LICS.2001.932499. URL https://www.cs.cmu.edu/~fp/papers/lics01.pdf.

[93] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Mathematical Foundations of Programming Semantics*, pages 209–228, New York, NY, 1990. Springer-Verlag. ISBN 978-0-387-34808-7. doi: 10.1007/BFb0040259.

[94] Brigitte Pientka, Andreas Abel, Francisco Ferreira, David Thibodeau, and Rébecca Zucchini. A type theory for defining logics and proofs. In *Proceedings of the 34th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, 2019.

[95] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Transactions on Programming Languages and Systems*, 22(1):1–44, January 2000. ISSN 0164-0925. doi: 10.1145/345099.345100. URL https://dl.acm.org/doi/10.1145/345099.345100.

[96] Andrew M. Pitts. Relational properties of domains. *Inf. Comput.*, 127(2):66–90, 1996. doi: 10.1006/INCO.1996.0052. URL https://doi.org/10.1006/inco.1996.0052.

[97] Bernhard Reus. *Program Verification in Synthetic Domain Theory*. Shaker Verlag Aachen, 1995.

[98] Giuseppe Rosolini and G. (Giuseppe. Continuity and effectiveness in topoi /. January 1986.

[99] Michael Shulman. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science*, 28(6):856–941, 2018. doi: 10.1017/S0960129517000147. URL https://doi.org/10.1017/S0960129517000147.

[100] A. Simpson. Computational adequacy for recursive types in models of intuitionistic set theory. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 287–298, 2002. doi: 10.1109/LICS.2002.1029837.

[101] Philipp Stassen, Daniel Gratzer, and Lars Birkedal. {mitten}: A Flexible Multimodal Proof Assistant. In *DROPS-IDN/v2/document/10.4230/LIPIcs.TYPES.2022.6*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPIcs.TYPES.2022.6. URL https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.TYPES.2022.6.

[102] Jonathan Sterling, Daniel Gratzer, and Lars Birkedal. Denotational semantics of general store and polymorphism. *CoRR*, abs/2210.02169, 2022. doi: 10.48550/ARXIV.2210.02169. URL https://doi.org/10.48550/arXiv.2210.02169.

[103] Jonathan Sterling, Daniel Gratzer, and Lars Birkedal. Free theorems from univalent reference types. *CoRR*, abs/2307.16608, 2023. doi: 10.48550/ARXIV.2307.16608. URL https://doi.org/10.48550/arXiv.2307.16608.

[104] The Agda Team. Agda, 2022. URL https://agda.readthedocs.io/en/latest/language/guarded-cubical.html.

[105] Hermann Thorisson. *Coupling, stationarity, and regeneration*. Probability and its Applications (New York). Springer-Verlag, New York, 2000. ISBN 0-387-98779-7.

[106] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. URL https://homotopytypetheory.org/book.

[107] Matthijs Vákár, Ohad Kammar, and Sam Staton. A domain theory for statistical probabilistic programming. *Proc. ACM Program. Lang.*, 3(POPL):36:1–36:29, 2019. doi: 10.1145/3290349. URL https://doi.org/10.1145/3290349.

[108] Niccolò Veltri and Andrea Vezzosi. Formalizing π-calculus in guarded cubical agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 270–283, 2020.

[109] Vezzosi, Andrea. `agda-flat`, 2018. URL https://github.com/agda/agda/tree/flat.

[110] C. Villani. *Optimal Transport: Old and New*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 2008. ISBN 9783540710509.

[111] Mitchell Wand, Ryan Culpepper, Theophilos Giannakopoulos, and Andrew Cobb. Contextual equivalence for a probabilistic language with continuous random variables and recursion. *Proc. ACM Program. Lang.*, 2(ICFP):87:1–87:30, 2018. doi: 10.1145/3236782. URL https://doi.org/10.1145/3236782.

[112] Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. Interaction trees: representing recursive and impure programs in coq. *Proc. ACM Program. Lang.*, 4(POPL):51:1–51:32, 2020. doi: 10.1145/3371119. URL https://doi.org/10.1145/3371119.

[113] Yizhou Zhang and Nada Amin. Reasoning about "reasoning about reasoning": semantics and contextual equivalence for probabilistic programs with nested queries and recursion. *Proc. ACM Program. Lang.*, 6(POPL):1–28, 2022. doi: 10.1145/3498677. URL https://doi.org/10.1145/3498677.