THESIS ON INFORMATICS AND SYSTEM ENGINEERING C123

A Type-Theoretical Study of Nontermination

NICCOLÒ VELTRI

TTÜ PRESS

TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies
Department of Software Science

This dissertation was accepted for the defense of the degree of Doctor of Philosophy in Informatics on 8 March 2017.

Supervisors: Tarmo Uustalu, PhD
Tallinn University of Technology

James Chapman, PhD
University of Strathclyde, UK

Opponents: Venanzio Capretta, PhD
University of Nottingham, UK

Martin Hötzel Escardó, PhD
University of Birmingham, UK

Defense: 26 May 2017

Declaration: Hereby I declare that this doctoral thesis, my original investigation and achievement, submitted for the doctoral degree at Tallinn University of Technology has not been previously submitted for any degree or examination.

/Niccolò Veltri/

ISSN 1406-4731
ISBN 978-9949-83-100-5 (publication)
ISBN 978-9949-83-101-2 (PDF)

INFORMAATIKA JA SÜSTEEMITEHNIKA C123

Tüübiteoreetiline uurimus mittetermineeruvusest

NICCOLÒ VELTRI

TTÜ KIRJASTUS

Contents

List of publications

I Chapman, J., Uustalu, T., Veltri, N.: Quotienting the delay monad by weak bisimilarity. In: Leucker, M., Rueda, C., Valencia, F. D. (eds.) Proc. of 12th Int. Coll. on Theoretical Aspects of Computing, ICTAC 2015, Lect. Notes in Comput. Sci., v. 9399, pp. 110–125, Springer (2015)

II Veltri, N.: Two set-based implementations of quotients in type theory. In: Nummenmaa, J., Sievi-Korte, O., Mäkinen, E. (eds.) Proc. of 14th Symposium on Programming Languages and Software Tools, SPLST 2015, CEUR Workshop Proceedings, v. 1525, pp. 194–205, CEUR-WS.org (2015)

III Chapman, J., Uustalu, T., Veltri, N.: Formalizing restriction categories. J. Formalized Reasoning, 10(1), 1–36 (2017)

Author's contribution to the publications

The candidate is responsible for the development of the theory described in this thesis and their Agda formalization. In addition, the candidate had the lead role in conceiving, drafting and producing the manuscripts. Also the work was presented at conferences by the thesis author.

The idea of formalizing restriction categories in Agda originated from James Chapman. The general idea of proving that the quotiented delay monad is universal among monads for non-termination was conceptualized and motivated by Tarmo Uustalu. Both Chapman and Uustalu helped in the writing of the articles on which some parts of the thesis are based on.

Accompanying code

The thesis is accompanied with formalization in the Agda proof assistant [1, 74]. The code is located at:

- `http://cs.ioc.ee/~niccolo/thesis/`.

The development uses Agda version 2.5.2 and Agda Standard Library version 0.13.

Introduction

Motivation

Martin-Löf type theory is a formal system for the development of constructive mathematics and a very expressive functional programming language. In this language, it is possible to reason and prove properties about the implemented programs. For this reason, interactive theorem provers based on (variations of) Martin-Löf type theory, such as Coq, Agda or Lean, are very useful tools widely employed in software verification and security.

Being a foundational system for the development of mathematics, programs written in Martin-Löf type theory must be terminating. Several techniques for implementing possibly non-terminating programs have been developed. Notorious is Capretta's approach, in which possible non-termination is treated as a computational effect. Applications of Capretta's solution include representation of general recursive functions [26], formalization of domain theory [17], operational semantics for While languages [41] and normalization by evaluation [2].

While formal reasoning about non-termination is important, it is notoriously subtle and connected with inherent difficulties for foundational reasons. My thesis makes a contribution here by solving a number of technical problems concerning termination/non-termination reasoning in dependently typed programming. It furthers the principles and practice of the latter and thereby trustworthy software technology.

Problem statement

In this thesis, we present a formalization of the mathematics of partiality in dependently typed programming, specifically in Martin-Löf type theory. In particular, we continue the study of Capretta's delay monad [26]. This monad serves as a constructive alternative to the maybe monad, allowing the implementation of possibly non-terminating programs in type theory. An element in the delay datatype (that we typically refer to as a computation) is specified by two ingredients: its rate of convergence and, in case of termination, a value. Generally we are interested only in the terminating/non-terminating behavior of a computation. This requires us

to quotient the delay datatype by the weak bisimilarity relation. Two computations are (termination-sensitively) weakly bisimilar if, whenever the first converges to a value, the second also converges to the same value, and vice versa. We can say that the quotiented delay datatype is an useful tool for modeling partiality in type theory, or that the quotiented delay monad (if it is indeed a monad) introduces partiality as an effect in type theory. Our goal in this thesis is to make such statements rigorous. Our strategy to achieve the latter can be summarized as follows:

- we prove that the delay monad quotiented by weak similarity is a monad;
- we choose a well-established notion of "framework for partiality" from category theory and we formalize its theory in the Agda proof assistant [1, 74];
- inside this framework, we identify a good notion of "monad for partiality" and we prove that the quotiented delay monad is canonical among such monads.

Contribution of the thesis

The strategy sketched above led us to the following contributions:

- We check if the delay datatype quotiented by weak bisimilarity is indeed a monad. Different approaches to quotients in type theory gives different results. Type theory does not possess quotient types. Quotients can be simulated by working with setoids, i.e., pairs consisting of a set and an equivalence relation. Another possibility is to extend type theory with inductive-like quotients à la Martin Hofmann [54]. Using setoids, it is easy to show that the quotiented delay datatype is a monad. With inductive-like quotients, the answer is surprisingly more involved: we need to employ classical or semi-classical principles, notably the limited principle of omniscience, the axiom of (weak) countable choice or the axiom of propositional choice, in order to construct the monad structure. This exposes a more general foundational problem in type theory: a bad interaction between quotients and infinite datatypes such as non-wellfounded or wellfounded but infinitely branching trees. We formalize this development in Agda.
- We develop a general category-theoretical framework for reasoning about partiality in Agda. Our choice fell on Cockett and Lack's restriction categories [35]. These categories possess an operation on homsets called restriction, subject to some axioms. Intuitively, the

restriction of a function is the partial identity specifying the domain of definedness of the function. We present a formalization of the first chapters of the theory of restriction categories. In particular, our formalization includes the proof that every restriction category can be fully embedded in a partial map category.

- We prove that the quotiented delay monad is canonical among monads for non-termination from iteration (in the sense of repeat-until/tail-recursion). We introduce (and formalize in Agda) the notion of ω-complete pointed classifying monad, as a combination of the existing notions of classifying monad [36] and finite-join restriction category [53]: a ω-complete pointed classifying monad is a monad whose Kleisli category is a restriction category which is ω**CPPO**-enriched with respect to the ordering specified by the restriction operation. This class of monads captures the notion of monad for non-termination from iteration. We show that the quotiented delay monad is the initial ω-complete pointed classifying monad in type theory.

Outline of the thesis

In Chapter 1, we present background material. This includes treatment of partiality in category theory and Martin-Löf type theory. Moreover we review some categorical approaches to iteration and other "cyclic" phenomena.

In Chapter 2, we review some aspects of Martin-Löf type theory, the formal system in which we are developing our mathematics. We describe the basics of the system and the additional principles that we need to postulate. We give a detailed exposition of two extensions of the theory that are of specific interest for our results: inductive-like quotient types and choice principles, notably (weak) countable choice and propositional choice. extensions are The material in this chapter appears in the author's publication Paper I [30]. The description of quotient types given in Section 2.2 also appears in Paper II [84].

In Chapter 3, we introduce Capretta's delay datatype. We know that the latter type defines a monad. The main goal of this chapter is to understand if the delay datatype quotiented by weak bisimilarity preserves the monad structure. We prove that this is the case if one postulates classical (the limited principle of omniscience) or semi-classical (weak countable choice and propositional choice) principles. Moreover, we show that the quotiented delay monad delivers ω-complete pointed partial orders, assuming countable choice. On top of this, we provide a new implementation of a monad for partiality in homotopy type theory and we compare the latter with the quotiented delay monad and Altenkirch et al.'s monad delivering free ω-

complete partial orders by construction [9]. The material in this chapter, apart from Sections 3.4 and 3.5, appears in the author's publication Paper I [30]. The material in Sections 3.4 and 3.5 is unpublished, but appears in the manuscript for a journal version of [30], currently under review. Section 3.2.3 on a solution using propositional choice was added for completeness sake after this work was completed; the new ideas belong to Martin Escardó.

In Chapter 4, we introduce the notion of ω-complete pointed classifying monad. In type theory, the initial such monad is given by the quotiented delay monad. This shows that the latter is canonical among monads for non-termination from iteration. In order to make the initiality result meaningful, we provide other non-trivial examples of ω-complete pointed classifying monads in type theory. Our main examples include partial map classifiers, constructed using Rosolini's notion of dominance. The material in this chapter is currently unpublished. It has been submitted to a conference.

In Appendix A, we describe a formalization of the mathematics of partiality in Agda. More specifically, we formalize the theory of restriction categories and their connection with partial map categories. This corresponds to a formalization of Section 4.1. We have developed our own library of basic utilities: implementation of quotient types, definitions of categories, functors, monics, isomorphisms, sections, idempotents and pullbacks; proofs of various properties about them, e.g., the pasting lemmas for pullbacks. The main part of the formalization consists of definitions of restriction categories, partial map categories; proofs of important lemmata; proof of the soundness Theorem 4.1; construction of splitting of idempotents; proof of the completeness Theorem 4.2. The material in this chapter appears in the author's publication Paper III [31].

The majority of the results presented in this thesis have been formalized in the Agda proof assistant. The formalization is available at `http://cs.ioc.ee/~niccolo/thesis/`.

Chapter 1

Background

In this chapter we discuss background material. As our aim is the study of Capretta's delay monad in Martin-Löf type theory, we give an overview of different ways to express partiality in category theory and type theory (Sections 1.1 and 1.2). Since we are also interested in a general treatment of non-termination in type theory, we review different categorical approaches to iteration (Section 1.3). In the next chapter, in particular Section 2.1, we give an overview of Martin-Löf type theory (we refer to [73] for a general introduction). For a general introduction to category theory we refer to [70, 10]. Alternatively, the reader can check our Agda formalization of restriction categories, that we discuss in Appendix A, which includes the majority of the basic definitions of category theory that we employ in this thesis.

1.1 Partiality in category theory

The simplest example of category of partial maps is given by the category **Pfn** of sets and partial functions, that can be presented in the following way:

Objects: sets.

Maps: a map $f : A \rightharpoonup B$ is a pair (A', f), where $A' \subseteq A$ and $f : A' \to B$.

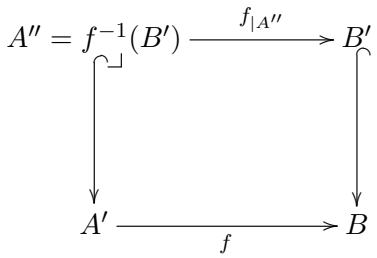
Identities: identity on A is the pair (A, id).

Composition: composition of maps $(A' \subseteq A, f : A' \to B)$ and $(B' \subseteq B, g : B' \to C)$ is given by the pair

$$A'' = f^{-1}(B') \subseteq A \qquad h = g \circ f_{|A''} : A'' \to C$$

The preimage of the function f, used to define the composition of partial

maps, can be constructed using the categorical notion of pullback:



The construction of **Pfn** can be generalized to what we refer to as partial map categories, that we formally introduce in Section 4.1. These are pairs consisting of a category $\mathbb{X}$ and a collection of monic morphisms in $\mathbb{X}$ satisfying a number of conditions. Such collections of monics appear under several different names in the literature: stable systems of monics [35], dominions [79, 47], admissible class of subobjects [77], notion of partial [78] and domain structure [72].

A different approach is given by providing direct, domain-free axiomatizations of categories of partial maps. This means that one does not build a category of partial morphisms on top of a given category of total morphisms, as one does in the definition of partial map categories. Instead, one requires from the start the existence of additional structure, sufficient for specifying a notion of partiality. Di Paola and Heller [42], motivated by the development of recursion theory in abstract categorical terms, introduced the notion of dominical category. The notion of partiality is specified by the existence of partial products and zero morphisms (maps which are nowhere defined). A refinement of this approach was given by Robinson and Rosolini [77]. They introduce the notion of p-category, in which the requirement of zero morphisms is dropped. Different axiomatics, but similar in flavor to p-categories, has been given by Carboni [28] and Curien and Obtulowicz [40].

Cockett and Lack introduced the notion of restriction category [35], which generalizes the earlier untyped axiomatization by Menger [69]. Restriction categories represent a minimalistic approach to partiality: the only requirement is the existence of a restriction operator satisfying certain equational conditions. The existence of a symmetric monoidal structure, fundamental in previous axiomatizations, is dropped. The restriction operator takes a morphism $f : A \to B$ to a morphism $\overline{f} : A \to A$, which has to be thought as specifying the domain of definedness of f. Intuitively, $\overline{f}$ is the partial function that behaves like identity on the elements on which f is defined. We review restriction categories and their connection with partial map categories in Section 4.1.

Often partial maps are the Kleisli maps of monad on the total map category. This is the situation of partial map classification [25, 36, 72].

Monads arising in this way in the restriction categories setting are called classifying monads. We discuss such monads Section 4.2.

Using restriction categories, categorical axiomatic approaches to computability [33] and complexity [34] have been developed. Moreover, restriction categories with additional structure (such as Cartesian products, meets, iteration, etc.) have been thoroughly investigated [37, 53]. Of particular interest for us are Guo's finite-join restriction categories [53]. Inspired by these, we introduce ω-complete pointed restriction categories. These are restriction categories which are ω**CPPO**-enriched wrt. the "less defined than" order on homsets induced by the restriction operation. In Chapter 4, we introduce the notion of ω-complete pointed classifying monad. These are monads whose Kleisli category is a ω-complete pointed restriction category in which pure maps are total.

The general categorical frameworks for partiality discussed in this section have found several applications, most notably in (axiomatic and synthetic) domain theory and semantics [47, 48, 57, 58]. Recently, a variation of join restriction categories has been used for modeling reversible recursion [11].

1.2 Partiality in type theory

In Martin-Löf type theory, described in Chapter 2, function types only contain total functions, therefore partial functions are not first-class objects. Nonetheless, there exist several techniques for encoding partiality in type theory. Recently, Bove et al. [23] have given a systematic overview of partiality and recursion in type theory and higher-order logic, describing also different tools for dealing with partial and recursive functions in theorem provers such as Agda, Coq and Isabelle/HOL. We recall some of the techniques specific for dealing with partiality in Martin-Löf type theory. In particular, we focus on techniques for representing partial functions without extending the logical foundation of the theory (that Bove et al. [23] call "definitional techniques"). This means that partial maps are considered as total maps in which either the domain is restricted to a particular subdomain or the codomain is extended with an undefined value. Classically, this corresponds to considering a partial function $f : X \rightharpoonup Y$ either as a total function of type $X' \to Y$, for some $X' \subseteq X$ (as in the presentation of the category **Pfn** given in the previous section), or as a total function of type $X \to Y + 1$, where the element of 1 represents undefinedness. In a constructive setting, the situation is more complex. The maybe monad $\mathsf{Maybe}\, X = X + 1$ cannot be used to model possibly non-terminating computations without the assumption of classical principles (more on this in Section 3.2.1).

Bove and Capretta [21, 20] showed that, if a partial function $f : X \rightharpoonup Y$ is given in terms of recursive equations, then it is possible to inductively

characterize the domain of definedness of f, i.e. a predicate $\mathsf{Dom}_f : X \to \mathcal{U}$, and represent f as a function of type $(\sum_{x:X} \mathsf{Dom}_f\, x) \to Y$. For nested recursive functions, the method uses inductive-recursive types [43]. Later, Bove [19] and Setzer [80] came up with similar methods that do not rely on inductive-recursive definitions.

Capretta [26] introduced the coinductive delay monad D as constructive alternative to the maybe monad. He also considered the delay monad quotiented by weak bisimilarity in the setoid approach. A partial function $f : X \rightharpoonup Y$ is given by a map of type $X \to \mathsf{D}\, Y$. Megacz [68] presented a refined version of Capretta's datatype. The delay monad is formally introduced in Section 3.1. Bove and Capretta [22] described a method that gives a coinductive characterization of the codomain of functions given in terms of recursive equations, dualizing the method described in [21, 20] for inductively specifying the domain of definedness.

Altenkirch et al. [9] have defined a monad which delivers ω-complete partial orders by construction. They work in homotopy type theory, and their construction makes use of a particular form of higher inductive types, called higher inductive-inductive types. We discuss how their monad relates to our constructions in Section 3.5.

Very recently, Escardó and Knapp [46] have introduced partial functions via dominances in homotopy type theory. We give a general account on dominances and their connection to ω-complete pointed classifying monads in Section 4.5.1.

Another approach consists in formalizing a categorical framework for partiality in type theory. We gave an overview of such frameworks in the previous section. This approach is more general and flexible: the notion of partiality is not restricted to undefinedness and non-termination, but takes into account a larger spectrum of other possibilities. Benton et al.'s formalization of domain theory in Coq [17] is an example of this approach. In this thesis, we also report on our formalization of the theory of Cockett and Lack's restriction categories [35] in type theory, described in Appendix A.

1.3 Categorical approaches to iteration

In Section 1.1, we described several general categorical settings for partiality. In this section we present categorical frameworks specific for modeling iteration and other cyclic phenomena.

Joyal et al. [60] introduced the notion of traced monoidal category, i.e. a symmetric monoidal category with a feedback operation

$$\frac{f : A \otimes X \to B \otimes X}{\mathsf{Tr}\, f : A \to B}$$

satisfying a number of conditions. When the monoidal structure is given by finite coproducts, one can interpret the feedback as a generic repeat-until loop. In fact, giving a feedback is equivalent to giving an iteration operation

$$\frac{f : X \to A + X}{\mathsf{iter}\, f : X \to A}$$

satisfying a number of conditions [29]. The most important of these conditions, which characterizes the behavior of iteration, is the unfolding axiom:

$$\mathsf{iter}\, f = [\mathsf{id}_A, \mathsf{iter}\, f] \circ f.$$

Dually, when the monoidal structure is given by finite products, one can interpret the feedback as a parametric fixpoint operation [56, 81]. Building on these results, Milius and Litak [70] have given an axiomatic formulation of guarded fixpoint operations.

A different but closely related direction of research investigates iteration as an effect, following Moggi's idea of using monads for modeling effectful computations [71]. Aczel et al. [4, 6] studied the notion of completely iterative monad. These monads provide unique solutions to guarded systems of recursive equations and constitute the categorical parallel of Elgot's completely iterative theories [44].

Solution to unguarded systems of recursive equations can be constructed using complete Elgot monads [51, 50, 52]. These are monads whose Kleisli category possesses an iteration operator with the same type of the operator iter described above, satisfying analogous conditions.

Chapter 2

Type theory

In this chapter, we give an overview of Martin-Löf type theory, a foundational system for constructive mathematics developed by the Swedish logician Per Martin-Löf. We summarize its main features and we specify our extensions to the system. For a general introduction to Martin-Löf type theory we refer to [73]. In Section 2.1, we give a short presentation of the theory. In Section 2.2, we describe how to represent quotients in type theory. In Section 2.3, we discuss the axiom of choice, countable choice, weak countable choice and propositional choice.

2.1 Martin-Löf type theory

We consider Martin-Löf type theory with inductive and coinductive types and a cumulative hierarchy of universes $\mathcal{U}_k$. The first universe is simply denoted $\mathcal{U}$ and when we write statements like "X is a type", we mean $X : \mathcal{U}$ unless otherwise specified. We write $=$ for judgmental (definitional) equality.

Given a type $X : \mathcal{U}$ and a type family $Y : X \to \mathcal{U}$, we indicate with $\prod_{x:X} Y\,x$ the type of *dependent functions* between X and Y. The elements of a dependent function type are functions whose codomain type can vary depending on the element of the domain to which the function is applied. In other words, $f : \prod_{x:X} Y\,x$ is a function that, given $x : X$, returns an element $f\,x : Y\,x$. A dependent function type can be interpreted set-theoretically as an indexed product over a given type (hence the resembling notation). By considering a constant type family (i.e., a type) $Y : \mathcal{U}$, we obtain the usual non-dependent function space $X \to Y$. We allow dependent functions to have implicit arguments and indicate implicit argument positions with curly brackets (as in Agda [1, 74]).

Given a type $X : \mathcal{U}$ and a type family $Y : X \to \mathcal{U}$, we indicate with $\sum_{x:X} Y\,x$ the type of *dependent pairs* of X and Y. The dependent pair type is a generalization of the product type, in which we allow the type of the second component of a pair to vary depending on the choice of the first

component. In other words, every element $p : \sum_{x:X} Y\,x$ is a pair $p = (x, y)$, with $x : X$ and $y : Y\,x$. A dependent pair type can be interpreted set-theoretically as an indexed sum over a given type (hence the resembling notation). By considering a constant type family $Y : \mathcal{U}$, we obtain the usual non-dependent product $X \times Y$. We write fst and snd for the first and second projection of a pair:

$$\mathsf{fst}\,(x, y) = x \quad \mathsf{snd}\,(x, y) = y$$

We specify *inductive types* using single rule lines. For example, the type of natural numbers $\mathbb{N}$ is defined by the rules

$$\frac{}{\mathsf{zero} : \mathbb{N}} \quad \frac{n : \mathbb{N}}{\mathsf{suc}\,n : \mathbb{N}}$$

We specify *coinductive types* using double rule lines. For example, the type of conatural numbers $\mathbb{N}^\infty$ is defined by the rules

$$\frac{}{\mathsf{zero} : \mathbb{N}^\infty} \quad \frac{n : \mathbb{N}^\infty}{\mathsf{suc}\,n : \mathbb{N}^\infty}$$

The double line signifies that the rule can be potentially applied an infinite amount of times. For example, applying infinitely often the second constructor of $\mathbb{N}^\infty$ one can corecursively define $\infty = \mathsf{suc}\,\infty$.

We indicate with 1 the *unit type*, inductively defined by the rule

$$\frac{}{* : 1}$$

We indicate with 0 the *empty type* with no constructors. The *disjoint union* (or coproduct) $X + Y$ of two types X and Y is inductively defined by the rules

$$\frac{x : X}{\mathsf{inl}\,x : X + Y} \quad \frac{y : Y}{\mathsf{inr}\,y : X + Y}$$

A fundamental example of inductive type is the *identity type*. Here we define the identity type as *heterogeneous* propositional equality. For all $x : X$ and $y : Y$, the type $x \equiv y$ is inductively defined by the rule

$$\frac{}{\mathsf{refl} : x \equiv x}$$

Symmetry, transitivity and substitutivity of $\equiv$ are named sym, trans and subst, respectively. We use a heterogeneous equality instead of a homogeneous one for readability and usability reasons. In particular, we want to avoid excessive usage of the substitutivity proof subst.

2.1.1 Additional principles

On top of basic Martin-Löf type theory we postulate a series of principles. We assume the principle of *function extensionality*, expressing that pointwise equal functions are equal, i.e., the inhabitedness of

$$\mathsf{FunExt} = \prod_{\{X,Y:\mathcal{U}\}} \prod_{\{f_1,f_2:X\to Y\}} \left(\prod_{x:X} f_1\, x \equiv f_2\, x\right) \to f_1 \equiv f_2$$

We assume function extensionality also for dependent functions. Likewise we will assume analogous extensionality principles stating that strongly bisimilar coinductive data and proofs are equal for the relevant coinductive types and predicates, namely, the delay datatype and weak bisimilarity (check DExt and $\approx\mathsf{Ext}$ in Section 3.1).

We also assume *uniqueness of identity proofs* for all types, i.e., an inhabitant for

$$\mathsf{UIP} = \prod_{\{X:\mathcal{U}\}} \prod_{\{x_1,x_2:X\}} \prod_{p_1,p_2:x_1\equiv x_2} p_1 \equiv p_2.$$

A type X is said to be a *proposition*, if it has at most one inhabitant, i.e., if the type

$$\mathsf{isProp}\, X = \prod_{x_1,x_2:X} x_1 \equiv x_2$$

is inhabited. We say that a predicate $P : X \to \mathcal{U}$ is *propositional* if $\mathsf{isProp}\,(P\,x)$ holds for all $x : X$.

For propositions, we postulate a further and less standard principle of *proposition extensionality*, stating that logically equivalent propositions are equal:

$$\mathsf{PropExt} = \prod_{\{X,Y:\mathcal{U}\}} \mathsf{isProp}\, X \to \mathsf{isProp}\, Y \to (X \leftrightarrow Y) \to X \equiv Y$$

Here $X \leftrightarrow Y = (X \to Y) \times (Y \to X)$.

Alternatively, we could set our development in *homotopy type theory* [83], but restrict ourselves to work with 0-truncated types, i.e., sets. In the latter framework, the principles FunExt and $\mathsf{PropExt}$ are consequences of the univalence axiom, while the restriction to 0-truncated types implies UIP.

2.2 Quotient types

Martin-Löf type theory does not have built-in quotient types. There are two main approaches to compensate for this: to mimic them by working with setoids or to extend the type theory with inductive-like quotients à la Hofmann.

2.2.1 Setoids

A *setoid* is a type X equipped with an equivalence relation R on X [13]. The equivalence relation R has to be thought of as a particular notion of "equality" on X. So one can think of the setoid (X, R) as the quotient of X by R. Quotienting a setoid (X, R) further corresponds to replacing the equivalence relation R with a coarser one.

The notion of setoid is very useful, but has some shortcomings. When working with setoids, every type comes with an equivalence relation (an idea that can be traced back to Bishop [18]). This approach forces the change of the concept of function type. A *setoid morphism* between setoids (X, R) and (Y, S) is a function $f : X \to Y$ such that $x_1 R x_2$ implies $(f x_1) S (f x_2)$, for all $x_1, x_2 : X$. In other words, a setoid morphism is a function that sends "equal" values in the domain into "equal" values in the codomain. So every time we define a function between two types, we also have to provide an additional proof of compatibility with the equivalence relation.

Another drawback appears when defining types depending on other types, e.g. the type of lists over a type X. Now we have to construct setoid-lists over a setoid (X, R). The carrier type is the usual type of lists inductively defined by the rules

$$\frac{}{[] : \mathsf{List}\, X} \qquad \frac{x : X \quad xs : \mathsf{List}\, X}{x :: xs : \mathsf{List}\, X}$$

We also have to lift the equivalence relation R to an equivalence relation $\mathsf{List}\, R$ on $\mathsf{List}\, X$. The relation $\mathsf{List}\, R$ is the functorial lifting of R and is inductively defined by the rules

$$\frac{}{[]\,(\mathsf{List}\, R)\,[]} \qquad \frac{x_1 R x_2 \quad xs_1\, (\mathsf{List}\, R)\, xs_2}{x_1 :: xs_1\, (\mathsf{List}\, R)\, x_2 :: xs_2}$$

Notice that this operation must be performed for all polymorphic type formers. We want to avoid this so-called "setoid hell". Therefore we choose to work with quotient types, introduced in the next section.

2.2.2 Inductive-like quotients

In this section, we describe quotient types as particular inductive-like types introduced by M. Hofmann in his PhD thesis [54]. These quotient types are ordinary types rather than setoids. The particular specification of quotient types described below has been given in [84], where we also discussed an impredicative encoding of quotients, reminiscent of Church numerals and more generally of encodings of inductive types in Calculus of Constructions.

Let X be a type and R an equivalence relation on X. A *quotient* of X by R is described by the following data (i)–(iv):

(i) a carrier type Q;

(ii) a constructor $[\_] : X \to Q$;

For any family of types $Y : Q \to \mathcal{U}$ and dependent function $f : \prod_{x:X} Y\,[x]$, we say that f is *R-compatible* (or simply *compatible*, when the intended equivalence relation is clear from the context), if the type

$$\mathsf{compat}\, f = \prod_{\{x_1,x_2:X\}} x_1 R x_2 \to f\, x_1 \equiv f\, x_2$$

is inhabited.

(iii) a proof $\mathsf{sound} : \mathsf{compat}\,[\_]$;

(iv) a dependent eliminator: for every family of types $Y : Q \to \mathcal{U}$ and function $f : \prod_{x:X} Y\,[x]$ with $p : \mathsf{compat}\, f$, there exists a function $\mathsf{lift}\, f\, p : \prod_{q:Q} Y\, q$ together with a computation rule

$$\mathsf{lift}_\beta\, f\, p\, x : \mathsf{lift}\, f\, p\,[x] \equiv f\, x$$

for all $x : X$.

Let $\mathsf{Quotient}\, X\, R : \mathcal{U}_1$ be the type of all such records. The type $\mathsf{Quotient}\, X\, R$ is not a proposition, since the carrier of a quotient is unique only up to isomorphism. We typically write X/R instead of Q for the carrier of a quotient of X by R.

We postulate the inhabitedness of $\mathsf{Quotient}\, X\, R$ for all types X and equivalence relations R on X. Hofmann has shown that quotient types are a conservative extension of Martin-Löf type theory [54]. We also postulate the existence of a *large* dependent eliminator, i.e., one that takes as an argument a family of types of the form $Y : X/R \to \mathcal{U}_1$.

The *propositional truncation* (or *squash*) $\|X\|$ of a type X is the quotient of X by the total relation $x_1 \mathsf{Total}\, x_2 = 1$, i.e., an inhabitant of $\mathsf{Quotient}\, X\, \mathsf{Total}$. We write $|\_|$ instead of $[\_]$ for the constructor of $\|X\|$. The non-dependent version of the elimination principle of $\|X\|$ is employed several times in this thesis, so we spell it out: in order to construct a function of type $\|X\| \to Y$, one has to construct a constant function of type $X \to Y$. The type $\|X\|$ can have at most one inhabitant, informally, an "uninformative" proof of X. For example, an inhabitant of $\|\sum_{x:X} P\, x\|$ can be thought of as a proof of there existing an element of X that satisfies P from which all information has been removed: both the witness element and the proof that it is good. Propositional truncation and other notions of weak or anonymous existence have been thoroughly studied in type theory [63].

We call a function $f : X \to Y$ *surjective*, and we write $\mathsf{isSurj}\, f$, if the type $\prod_{y:Y} \|\sum_{x:X} f\, x \equiv y\|$ is inhabited. We call f a *split epimorphism*, and we write $\mathsf{isSplitEpi}\, f$, if the type $\|\sum_{g:Y\to X} \prod_{y:Y} f\,(g\, y) \equiv y\|$ is inhabited. We say that f is a *retraction*, if the type $\sum_{g:Y\to X} \prod_{y:Y} f\,(g\, y) \equiv y$ is inhabited. Every retraction is a split epimorphism, and every split epimorphism is surjective.

Proposition 2.1. *The constructor* $[\_]$ *is surjective for all quotients.*

Proof. Given a type X and an equivalence relation R on X, we define:

$$[\_]\mathsf{surj} : \prod_{q:X/R} \left\| \sum_{x:X} [x] \equiv q \right\|$$
$$[\_]\mathsf{surj} = \mathsf{lift}\,(\lambda\, x.\, |x, \mathsf{refl}|)\, p$$

The compatibility proof p is trivial, since $|x_1, \mathsf{refl}| \equiv |x_2, \mathsf{refl}|$ for all $x_1, x_2 : X$. □

A quotient X/R is said to be *effective*, if the type $\prod_{x_1,x_2:X} [x_1] \equiv [x_2] \to x_1\, R\, x_2$ is inhabited. In general, effectiveness does not hold for all quotients. In fact, postulating effectiveness for all quotients implies LEM [66]. But we can prove that all quotients satisfy a weaker property. We say that a quotient X/R is *weakly effective*, if the type $\prod_{x_1,x_2:X} [x_1] \equiv [x_2] \to \|x_1 R x_2\|$ is inhabited.

Proposition 2.2. *All quotients are weakly effective.*

Proof. Let X be a type, R an equivalence relation on X and $x : X$. Consider the function $\|x\ R\ \_\| : X \to \mathcal{U}$, $\|x\ R\ \_\| = \lambda\, x'.\, \|x\ R\ x'\|$. We show that $\|x\ R\ \_\|$ is R-compatible. Let $x_1, x_2 : X$ with $x_1\ R\, x_2$. We have $x\ R\, x_1 \leftrightarrow x\ R\, x_2$ and therefore $\|x\ R\, x_1\| \leftrightarrow \|x\ R\, x_2\|$. Since propositional truncations are propositions, using proposition extensionality, we conclude $\|xRx_1\| \equiv \|xRx_2\|$. We have constructed a term $p_x : \mathsf{compat}\, \|x\ R\ \_\|$, and therefore a function $\mathsf{lift}\, \|x\ R\ \_\|\, p_x : X/R \to \mathcal{U}$ (large elimination is fundamental in order to apply lift, since $\|x\ R\ \_\| : X \to \mathcal{U}$ and $X \to \mathcal{U} : \mathcal{U}_1$). Moreover, $\mathsf{lift}\, \|x\ R\ \_\|\, p_x\, [y] \equiv \|x\ R\ y\|$ by its computation rule.

Let $[x_1] \equiv [x_2]$ for some $x_1, x_2 : X$. We have:

$$\|x_1\ R\, x_2\| \equiv \mathsf{lift}\, \|x_1 R\ \_\|\, p_{x_1}\, [x_2] \equiv \mathsf{lift}\, \|x_1\ R\ \_\|\, p_{x_1}\, [x_1] \equiv \|x_1\ R\, x_1\|$$

and $x_1\ R\, x_1$ holds, since R is reflexive. □

As already pointed out, not all quotients are effective. This is because a relation R is generally proof-relevant, i.e., the type $x_1 R x_2$ is not a proposition, and there is no canonical way of constructing an inhabitant of $x_1 R x_2$

knowing $[x_1] \equiv [x_2]$ (remember that the latter type is a proposition, since we are assuming uniqueness of identity proofs). But notice that, if R is propositional, then the quotient X/R is effective, because a function of type $\|x_1 R x_2\| \to x_1 R x_2$ can be defined by lifting the identity function on $x_1 R x_2$, which is constant in this case. Moreover, for all equivalence relations R, the type X/R is also the carrier of an effective quotient of X by the equivalence relation $x_1 \| R \| x_2 = \|x_1 R x_2\|$.

Notice that the constructor $[\_]$ is not a split epimorphism for all quotients. The existence of a choice of representative for each equivalence class is a non-constructive principle, since it implies the *law of excluded middle*, i.e., the inhabitedness of the following type:

$$\mathsf{LEM} = \prod_{\{X:\mathcal{U}\}} \mathsf{isProp}\, X \to X + \neg X$$

where $\neg X = X \to 0$.

Proposition 2.3. *Suppose that* $[\_]$ *is a split epimorphism for all quotients. Then* LEM *is inhabited.*

Proof. Let X be a type together with a proof of $\mathsf{isProp}\, X$. We consider the equivalence relation R on Bool, $x_1 R x_2 = x_1 \equiv x_2 + X$. By hypothesis we obtain $\| \sum_{\mathsf{rep}:\mathsf{Bool}/R \to \mathsf{Bool}} \prod_{q:\mathsf{Bool}/R} [\mathsf{rep}\, q] \equiv q \|$. Using the elimination principle of propositional truncation, it is sufficient to construct a constant function of type

$$\left(\sum_{\mathsf{rep}:\mathsf{Bool}/R \to \mathsf{Bool}} \prod_{q:\mathsf{Bool}/R} [\mathsf{rep}\, q] \equiv q \right) \to X + \neg X$$

Let $\mathsf{rep} : \mathsf{Bool}/R \to \mathsf{Bool}$ with $[\mathsf{rep}\, q] \equiv q$ for all $q : \mathsf{Bool}/R$. We have $[\mathsf{rep}\, [x]] \equiv [x]$ for all $x : \mathsf{Bool}$, which by Proposition 2.2 implies $\|\mathsf{rep}\, [x]\, R\, x\|$.

Note now that the following implication (a particular instance of axiom of choice on Bool) holds:

$$\mathsf{acBool} : \left(\prod_{x:\mathsf{Bool}} \|\mathsf{rep}\, [x]\, R\, x\| \right) \to \left\| \prod_{x:\mathsf{Bool}} \mathsf{rep}\, [x]\, R\, x \right\|$$
$$\mathsf{acBool}\, r = \mathsf{lift}_2\, (\lambda\, r_1\, r_2.\, |\lambda\, x.\, \mathsf{if}\, x\, \mathsf{then}\, r_1\, \mathsf{else}\, r_2|)\, p\, (r\, \mathsf{true})\, (r\, \mathsf{false})$$

where $\mathsf{if}\, \mathsf{true}\, \mathsf{then}\, r_1\, \mathsf{else}\, r_2 = r_1$ and $\mathsf{if}\, \mathsf{false}\, \mathsf{then}\, r_1\, \mathsf{else}\, r_2 = r_2$, and lift_2 is the two-argument version of lift. The compatibility proof p is immediate, since the return type is a proposition.

We now construct a function of type $\|\prod_{x:\mathsf{Bool}} \mathsf{rep}\, [x]\, R\, x\| \to X + \neg X$. It is sufficient to define a function $(\prod_{x:\mathsf{Bool}} \mathsf{rep}\, [x]\, R\, x) \to X + \neg X$ (it will be constant, since the type $X + \neg X$ is a proposition, if X is a proposition), so

we suppose $\mathsf{rep}\,[x]\,R\,x$ for all $x : \mathsf{Bool}$. We analyze the (decidable) equality $\mathsf{rep}\,[\mathsf{true}] \equiv \mathsf{rep}\,[\mathsf{false}]$ on Bool. If it holds, then we have $\mathsf{true}\ R\ \mathsf{false}$ and therefore an inhabitant of X. If it does not hold, we have an inhabitant of $\neg X$: indeed, suppose $x : X$, then $\mathsf{true}\,R\,\mathsf{false}$, so $[\mathsf{true}] \equiv [\mathsf{false}]$ and therefore $\mathsf{rep}\,[\mathsf{true}] \equiv \mathsf{rep}\,[\mathsf{false}]$ holds, which contradicts the hypothesis. □

One can always specify a quotient structure on the codomain of a surjective function $b : X \to Q$. We can define a relation $\equiv_b$ on X: $x_1 \equiv_b x_2$ if and only if $b\,x_1 \equiv b\,x_2$.

Proposition 2.4. *The type Q is the carrier of a quotient of X by the equivalence relation $\equiv_b$. The map b sends each element to its equivalence class.*

Proof. We construct a dependent eliminator lift. Given a family of types $Y : Q \to \mathcal{U}$, a $\equiv_b$-compatible map $f : \prod_{x:X} Y\,(b\,x)$ and an element $q : Q$. We have to give an inhabitant of $Y\,q$. We apply the proof of surjectivity of b to q, and we obtain a proof of $\| \sum_{x:X}\ b\,x \equiv q \|$. We are done using the elimination principle of propositional truncation and constructing a constant map $f' : (\sum_{x:X}\ b\,x \equiv q) \to Y\,q$.

$$f' : \left(\sum_{x:X} b\,x \equiv q\right) \to Y\,q$$
$$f'\,(x, e) = \mathsf{subst}\,Y\,e\,(f\,x)$$

Using the $\equiv_b$-compatibility of f, it is easy to see that f' is constant. □

Notice that the quotient described above is effective, since the relation $\equiv_b$ is propositional. Proposition 2.4 gives a sufficient condition to prove that a type is a quotient. This will be exploited in Sections 3.2.2 and 3.4.2.

2.3 Choice principles

In this section, we discuss the axiom of choice and two specific instances of it, the axiom of countable choice and the axiom of weak countable choice. We show why we cannot assume the full axiom of choice from a constructive perspective. The axiom of (weak) countable choice is a fundamental ingredient in proving that the delay datatype quotiented by weak bisimilarity is a monad. Moreover, it is necessary to show that the quotiented delay datatype delivers free ωcppos.

2.3.1 Axiom of choice

The *full axiom of choice* (AC) is defined as follows:

$$\mathsf{AC} = \prod_{\{X,Y:\mathcal{U}\}} \prod_{P:X\to Y\to\mathcal{U}} \left(\prod_{x:X} \left\| \sum_{y:Y} P\,x\,y \right\|\right) \to \left\| \sum_{f:X\to Y} \prod_{x:X} P\,x\,(f\,x) \right\|$$

Notice that AC is fundamentally different from the *type-theoretic* axiom of choice:

$$\prod_{\{X,Y:\mathcal{U}\}} \prod_{P:X\to Y\to\mathcal{U}} \left(\prod_{x:X} \sum_{y:Y} P\,x\,y\right) \to \sum_{f:X\to Y} \prod_{x:X} P\,x\,(f\,x)$$

which is provable in type theory.

AC is a controversial semi-classical axiom, generally not accepted in constructive systems [67]. We give some alternative formulation of AC.

Theorem 2.1. *The following are logically equivalent:*

(i) AC*;*

(ii) $\mathsf{AC}_2 = \prod_{\{X:\mathcal{U}\}} \prod_{P:X\to\mathcal{U}} (\prod_{x:X} \|P\,x\|) \to \|\prod_{x:X} P\,x\|$*;*

(iii) every surjective map is a split epimorphism;

(iv) the covariant hom-functor $Z \to \_ : (X \to Y) \to ((Z \to X) \to (Z \to Y))$*,* $Z \to g = \lambda f.\, g \circ f$*, preserves surjections;*

(v) the map $Z \to [\_] : (Z \to X) \to (Z \to X/R)$ *is surjective for all quotients* X/R*.*

We do not give a proof of Theorem 2.1. We will prove Theorem 2.2 instead, which has a similar proof.

We reject the axiom of choice, since in our system it implies the law of excluded middle.

Proposition 2.5. AC *implies* LEM*.*

Proof. Assume AC. By Theorem 2.1, $(i)\to(iii)$, we have that the surjective constructor $[\_]$ is a split epimorphism for all quotients X/R. By Proposition 2.3, this implies LEM. □

2.3.2 Axiom of countable choice

The *axiom of countable choice* ($\mathsf{AC}\omega$) is a specific instance of the axiom of choice where the binary predicate P has its first argument in $\mathbb{N}$:

$$\mathsf{AC}\omega = \prod_{\{X:\mathcal{U}\}} \prod_{P:\mathbb{N}\to X\to\mathcal{U}} \left(\prod_{n:\mathbb{N}} \left\|\sum_{x:X} P\,n\,x\right\|\right) \to \left\|\sum_{f:\mathbb{N}\to X} \prod_{n:\mathbb{N}} P\,n\,(f\,n)\right\|$$

Similarly to the full axiom of choice, countable choice is also a controversial principle [76]. But, differently from AC, it does not imply excluded middle and generally constructive mathematicians like it more [82, Ch. 4]. On the other hand, there exist models of type theory in which countable choice does not hold [39]. Countable choice can be given alternative formulations analogous to those given in Theorem 2.1 for AC.

Theorem 2.2. *The following are logically equivalent:*

(i) $\mathsf{AC}\omega$*;*

(ii) $\mathsf{AC}\omega_2 = \prod_{P:\mathbb{N}\to\mathcal{U}} (\prod_{n:\mathbb{N}} \|P\,n\|) \to \|\prod_{n:\mathbb{N}} P\,n\|$*;*

(iii) every surjective map of type $X \to \mathbb{N}$ *is a split epimorphism;*

(iv) the covariant hom-functor $\mathbb{N} \to \_ : (X \to Y) \to ((\mathbb{N} \to X) \to (\mathbb{N} \to Y))$ *preserves surjections;*

(v) the map $\mathbb{N} \to [\_] : (\mathbb{N} \to X) \to (\mathbb{N} \to X/R)$ *is surjective for all quotients* X/R*.*

Proof.

$(i)\to(ii)$: Assume $\mathsf{ac}\omega : \mathsf{AC}\omega$. Let $P : \mathbb{N} \to \mathcal{U}$ and $p : \prod_{n:\mathbb{N}} \|P\,n\|$. Define $P' : \mathbb{N} \to 1 \to \mathcal{U}$, $P'\,n\,* = P\,n$. Using p and the elimination principle of propositional truncation, it is easy to construct $p' : \prod_{n:\mathbb{N}} \|\sum_{*:1} P'\,n\,*\|$. By applying $\mathsf{ac}\omega$ to P' and p' we obtain an inhabitant of $\|\sum_{f:\mathbb{N}\to 1} \prod_{n:\mathbb{N}} P'\,n\,(f\,n)\|$, and from this we can conclude $\|\prod_{n:\mathbb{N}} P\,n\|$ using the elimination principle of propositional truncation.

$(ii)\to(iii)$: Assume $\mathsf{ac}\omega_2 : \mathsf{AC}\omega_2$. Let $f : X \to \mathbb{N}$ with a proof of surjectivity $p : \mathsf{isSurj}\,f$. Define $P\,n = \sum_{x:X} f\,x \equiv n$. By applying $\mathsf{ac}\omega_2$ to P and p we obtain an inhabitant of $\|\prod_{n:\mathbb{N}} P\,n\|$, and from this we can conclude $\mathsf{isSplitEpi}\,f$ using the elimination principle of propositional truncation.

$(iii)\to(iv)$: Suppose that every surjective function with codomain $\mathbb{N}$ is a split epimorphism. Let $g : X \to Y$ with a proof of surjectivity $p : \mathsf{isSurj}\,g$, let $h : \mathbb{N} \to Y$. Define $X' = \sum_{n:\mathbb{N}} \sum_{x:X} g\,x \equiv h\,n$ and

$h' : X' \to \mathbb{N}$, $h' = \mathsf{fst}$. Using the elimination principle of propositional truncation and the proof p, it is easy to prove the surjectivity of h'. Therefore h' is a split epimorphism, i.e., we have an inhabitant of $\| \sum_{s:\mathbb{N}\to X'} \prod_{n:\mathbb{N}} \mathsf{fst}\,(s\,n) \equiv n \|$. From this, using the elimination principle of propositional truncation, we can conclude $\| \sum_{f:\mathbb{N}\to X} g \circ f \equiv h \|$.

$(iv)\to(v)$: If the hom-functor $\mathbb{N} \to \_$ preserves surjections, than $\mathbb{N} \to [\_]$ is surjective, since the map $[\_]$ is surjective by Proposition 2.1.

$(v)\to(i)$: Suppose that the map $\mathbb{N} \to [\_]$ is surjective for all quotients. Let $P : \mathbb{N} \to X \to \mathcal{U}$ such that $p : \prod_{n:\mathbb{N}} \| \sum_{x:X} P\,n\,x \|$. Let $Y = \sum_{n:\mathbb{N}} \sum_{x:X} P\,n\,x$ and consider the following equivalence relation R on Y: $(n_1, x_1, q_1) R (n_2, x_2, q_2) = n_1 \equiv n_2$. Notice that the type $Q = \sum_{n:\mathbb{N}} \| \sum_{x:X} P\,n\,x \|$ is the carrier of a quotient of Y by R. The map $[\_] : Y \to Q$ sends a triple (n, x, q) to $(n, |x, q|)$. By hypothesis, the map $\mathbb{N} \to [\_]$ is surjective. We apply this surjectivity proof to the map $g : \mathbb{N} \to Q$, $g\,n = (n, p\,n)$. We obtain an inhabitant of $\| \sum_{f:\mathbb{N}\to Y} [\_] \circ f \equiv g \|$. Using the elimination principle of propositional truncation it is straightforward to conclude $\| \sum_{f:\mathbb{N}\to X} \prod_{n:\mathbb{N}} P\,n\,(f\,n) \|$.

□

We will assume $\mathsf{AC}\omega$ in Section 3.4 when proving that the quotiented delay datatype delivers free ωcppos. But we will see in Section 3.2.2 that, in order to prove that the quotiented delay datatype is a monad, it is sufficient to assume a weaker version of countable choice, that we introduce in the next section.

2.3.3 Axiom of weak countable choice

The *weak axiom of countable choice* ($\mathsf{WAC}\omega$) is a specific instance of the axiom of countable choice. We present it here as an instance of $\mathsf{AC}\omega_2$, introduced in point (ii) of Theorem 2.2:

$$\mathsf{WAC}\omega = \prod_{\{X:\mathcal{U}\}} \prod_{P:\mathbb{N}\to X\to\mathcal{U}} \left(\prod_{m<n} \mathsf{isProp} \left(\sum_{x:X} P\,m\,x \right) + \mathsf{isProp} \left(\sum_{x:X} P\,n\,x \right) \right)$$
$$\to \left(\prod_{n:\mathbb{N}} \left\| \sum_{x:X} P\,n\,x \right\| \right) \to \left\| \sum_{f:\mathbb{N}\to X} \prod_{n:\mathbb{N}} P\,n\,(f\,n) \right\|$$

The principle $\mathsf{WAC}\omega$ is like countable choice, but restricted to predicates $P : \mathbb{N} \to \mathcal{U}$ for which intuitively the type $\sum_{x:X} P\,n\,x$ is a proposition for all but possibly one $n : \mathbb{N}$. This is expressed by stating that, for any $m < n$, at

least one of $\sum_{x:X} P\,m\,x$ and $\sum_{x:X} P\,n\,x$ is a proposition. $\mathsf{WAC}\omega$ is an axiom with enough strength to prove Bishop's principle and the fundamental theorem of algebra [24].

We present three alternative formulations of $\mathsf{WAC}\omega$. We introduce some notation. Let $f : X \to Y$ and $y : Y$, we define the *preimage* of y under f as $f^{-1}y = \sum_{x:X} f\,x \equiv y$. We also introduce a variant of the covariant hom-functor $\mathbb{N} \to \_$. Given two types X, Y and a predicate $P : (\mathbb{N} \to Y) \to \mathcal{U}$, we define the map:

$$\mathbb{N} \to_P \_ : \prod_{g:X\to Y} \sum_{f:\mathbb{N}\to X} P\,(g \circ f) \to \sum_{f:\mathbb{N}\to Y} P\,f$$
$$(\mathbb{N} \to_P g)\,(f, p) = (g \circ f, p)$$

The function $\mathbb{N} \to_P g$ acts in the same way of $\mathbb{N} \to g$, but it only operates on maps $f : \mathbb{N} \to X$ such that $g \circ f$ satisfies the predicate P.

Theorem 2.3. *The following are logically equivalent:*

(i) $\mathsf{WAC}\omega$*;*

(ii)

$$\mathsf{WAC}\omega_2 = \prod_{P:\mathbb{N}\to\mathcal{U}} \left(\prod_{m<n} \mathsf{isProp}\,(P\,m) + \mathsf{isProp}\,(P\,n) \right) \to \left(\prod_{n:\mathbb{N}} \|P\,n\| \right) \to \left\| \prod_{n:\mathbb{N}} P\,n \right\|;$$

(iii) a surjective map $f : X \to \mathbb{N}$ is a split epimorphism if, for all $m < n$, we have $\mathsf{isProp}(f^{-1}\,m) + \mathsf{isProp}(f^{-1}\,n)$;

(iv) for any surjective $g : X \to Y$, we have that the map $\mathbb{N} \to_{P_g} g$ is surjective, where $P_g : (\mathbb{N} \to Y) \to \mathcal{U}$ is the following predicate:

$$P_g\,f = \prod_{m<n} \mathsf{isProp}\,(g^{-1}\,(f\,m)) + \mathsf{isProp}\,(g^{-1}\,(f\,n));$$

(v) the map $\mathbb{N} \to_P [\_]$ is surjective for all quotients, where $P : (\mathbb{N} \to X/R) \to \mathcal{U}$ is the following predicate:

$$P\,f = \prod_{m<n} \mathsf{isProp}\,([\_]^{-1}\,(f\,m)) + \mathsf{isProp}\,([\_]^{-1}\,(f\,n)).$$

Condition (v) states that the map $\mathbb{N} \to [\_]$ is surjective when restricted to streams $f : \mathbb{N} \to X$ for which the equivalence class of $f\,n$ is a proposition for all but possibly one $n : \mathbb{N}$. We refrain from proving Theorem 2.3 since its proof is a straightforward adaptation of the proof of Theorem 2.2.

The axiom of weak countable choice will be employed in Section 3.2.2 in order to prove that the quotiented delay datatype is a monad.

2.3.4 Axiom of propositional choice

The *axiom of propositional choice* is a variant of the full axiom of choice in which the binary predicate P has its first argument in a proposition X:

$$\mathsf{PAC} = \prod_{\{X,Y:\mathcal{U}\}} \prod_{P:X\to Y\to\mathcal{U}} \mathsf{isProp}\, X \\ \to \left(\prod_{x:X} \left\| \sum_{y:Y} P\, x\, y \right\|\right) \to \left\| \sum_{f:X\to Y} \prod_{x:X} P\, x\, (f\, x) \right\|$$

The axiom of propositional choice admits alternative formulations similar to the one of AC given in Theorem 2.1. Kraus et al. [63] show that PAC is equivalent to the inhabitedness of $\prod_{X:\mathcal{U}} \|\|X\| \to X\|$. PAC is known to fail in some toposes [49].

Following a suggestion by Martin Escardó, in Section 3.2.3 we will employ PAC to construct a monad structure on the quotiented delay datatype. We will use the following alternative formulation of PAC: given a proposition Z, a type X and an equivalence relation R on X, the following isomorphism holds:

$$(Z \to X)/(Z \to R) \cong (Z \to X/R)$$

where as usual $Z \to R$ is the pointwise lifting of R to the function space $Z \to X$.

2.4 Summary

In this chapter, we gave a brief description of Martin-Löf type theory, the constructive framework within we develop our results. A complete presentation of the framework can be found in [73]. The additional principles we introduced in Section 2.1.1 are standard extensionality principles that one often puts on top of the basic type theory in order to formalize and prove mathematical concepts [83].

Less standard is the quotient types extension presented in Section 2.2, inspired by Martin Hofmann's inductive-like quotients [54]. The material of Section 2.2 was extrapolated from the author's publications [30] and especially [84], in which the author also introduced an impredicative encoding of quotients reminiscent of Church numerals and more generally of encodings of inductive types in Calculus of Constructions.

In the last part of the chapter, we gave a brief introduction to choice principles we will exploit in several constructions. In particular, we presented alternative equivalent formulations of countable and weak countable choice. Countable choice is considered an important axiom of computability theory [14] and it is traditionally employed in the construction of Cauchy

real numbers. Weak countable choice is sufficient to prove Bishop's principle and the fundamental theorem of algebra [24]. The status of weak countable choice in Martin-Löf type theory is not very clear. It is known that there exist models of type theory in which countable choice does not hold [39].

Chapter 3

Delay monad

The delay datatype was introduced by Capretta [26] as a means to deal with partial functions (as in computability theory) in Martin-Löf type theory. It is used in this setting to cope with possible non-termination of computations (as, e.g., in the unbounded search of minimalization). Inhabitants of the delay datatype are delayed values, that we call computations. Crucially computations can be non-terminating and not return a value at all. The delay datatype constitutes a (strong) monad, which makes it possible to deal with possibly non-terminating computations just like any other flavor of effectful computations following Moggi's general monad-based method [71]. Often, one is only interested in termination of computations and not the exact computation time. Identifying computations that only differ by finite amounts of delay corresponds to quotienting the delay datatype by weak bisimilarity. The quotient datatype is used as a constructive alternative to the maybe datatype (see, e.g., [17]) and should also be a (strong) monad.

In Section 3.1, we present the delay datatype and (termination-sensitive) weak bisimilarity. In Section 3.2, we quotient the delay datatype by weak bisimilarity and we discuss how to construct a monad structure on top of it assuming one of the following three axioms: the limited principle of omniscience, weak countable choice or propositional choice. In Section 3.3, we look at the arrow structure (in the sense of Hughes [55]) on the Kleisli homsets for the delay monad and show how this survives quotienting by pointwise weak bisimilarity. In Section 3.4, we prove that the quotiented delay datatype delivers free ωcppos, assuming countable choice. In Section 3.5, we present a new monad for partiality in homotopy type theory and show how it relates to the quotiented delay monad and to Altenkirch et al.'s monad delivering free ωcppos by construction [9].

The results of this chapter have been formalized in the Agda proof assistant. The formalization is available at `http://cs.ioc.ee/~niccolo/thesis/`. The material presented in this chapter is based on a paper written by the author together with James Chapman and Tarmo Uustalu [30] and on a manuscript for a journal version of [30], currently under review.

3.1 Delay and weak bisimilarity

For a given type X, each element of the *delay type* $\mathsf{D}\,X$ is a possibly infinite computation that returns a value of X, if it terminates. We define $\mathsf{D}\,X$ as a coinductive type by the rules

$$\frac{}{\mathsf{now}\,x : \mathsf{D}\,X} \qquad \frac{c : \mathsf{D}\,X}{\mathsf{later}\,c : \mathsf{D}\,X}$$

The non-terminating computation never is corecursively defined as $\mathsf{never} = \mathsf{later}\,\mathsf{never}$.

Propositional equality is not suitable for coinductive types. We need different notions of equality, namely strong and weak bisimilarity.

Let R be an equivalence relation on a type X. The relation lifts to an equivalence relation $\sim_R$ on $\mathsf{D}\,X$ that we call *strong R-bisimilarity*. The relation is coinductively defined by the rules

$$\frac{p : x_1 R\, x_2}{\mathsf{now}_\sim\, p : \mathsf{now}\,x_1 \sim_R \mathsf{now}\,x_2} \qquad \frac{p : c_1 \sim_R c_2}{\mathsf{later}_\sim\, p : \mathsf{later}\,c_1 \sim_R \mathsf{later}\,c_2}$$

We alternatively denote the relation $\sim_R$ with $\mathsf{D}\,R$, since strong R-bisimilarity is the functorial lifting of the relation R to $\mathsf{D}\,X$. Strong $\equiv$-bisimilarity is simply called strong bisimilarity and denoted $\sim$.

While it ought to be the case, one cannot prove that strongly bisimilar computations are equal in intensional Martin-Löf type theory. Therefore we postulate an inhabitant for

$$\mathsf{DExt} = \prod_{\{X:\mathcal{U}\}} \prod_{\{c_1,c_2:\mathsf{D}\,X\}} c_1 \sim c_2 \to c_1 \equiv c_2$$

We take into account another equivalence relation $\approx_R$ on $\mathsf{D}\,X$ called *weak R-bisimilarity*, which is in turn defined in terms of *convergence*. The latter is a binary relation between $\mathsf{D}\,X$ and X relating terminating computations to their values. It is inductively defined by the rules

$$\frac{}{\mathsf{now}_\downarrow : \mathsf{now}\,x \downarrow x} \qquad \frac{p : c \downarrow x}{\mathsf{later}_\downarrow\, p : \mathsf{later}\,c \downarrow x}$$

Two computations are considered weakly R-bisimilar, if they differ by a finite number of applications of the constructor later (from where it follows classically that they either converge to R-related values or diverge). Weak R-bisimilarity is defined coinductively by the rules

$$\frac{p_1 : c_1 \downarrow x_1 \quad p_2 : x_1 R\, x_2 \quad p_3 : c_2 \downarrow x_2}{\downarrow_\approx p_1\, p_2\, p_3 : c_1 \approx_R c_2} \qquad \frac{p : c_1 \approx_R c_2}{\mathsf{later}_\approx\, p : \mathsf{later}\,c_1 \approx_R \mathsf{later}\,c_2}$$

Weak $\equiv$-bisimilarity is called just weak bisimilarity and denoted $\approx$. In this case, we modify the first constructor for simplicity:

$$\frac{p_1 : c_1 \downarrow x \quad p_2 : c_2 \downarrow x}{\downarrow_{\approx} p_1\, p_2 : c_1 \approx c_2}$$

Remark 3.1. Notice that the type $c_1 \approx c_2$ is not a proposition. But weak bisimilarity can be defined alternatively as the following propositional relation:

$$\frac{}{\mathsf{now}\, x \approx' \mathsf{now}\, x} \qquad \frac{c \downarrow x}{\mathsf{later}\, c \approx' \mathsf{now}\, x} \qquad \frac{c \downarrow x}{\mathsf{now}\, x \approx' \mathsf{later}\, c} \qquad \frac{c_1 \approx' c_2}{\mathsf{later}\, c_1 \approx' \mathsf{later}\, c_2}$$

We have $c \approx' c'$ if and only if $c \approx c'$. We prefer to work with $\approx$ instead of $\approx'$, since proofs of $\approx$ are somewhat easier to construct, there is more freedom.

In fact, there are even more robust versions of weak bisimilarity with even more proofs. For example, the following relation, closer to weak bisimilarity in CCS:

$$\frac{c_1 \downarrow x \quad c_2 \downarrow x}{c_1 \approx'' c_2} \qquad \frac{c_1 \searrow c_1' \quad c_1' \approx'' c_2' \quad c_2 \searrow c_2'}{\mathsf{later}\, c_1 \approx'' \mathsf{later}\, c_2}$$

where the type $c_1 \searrow c_2$ states that the computation c_1 reduces to the computation c_2 and it is inductively defined by the rules

$$\frac{}{c \searrow c} \qquad \frac{c_1 \searrow c_2}{\mathsf{later}\, c_1 \searrow c_2}$$

In this thesis we only use the relation $\approx$ and not its equivalent variants $\approx'$ and $\approx''$.

The delay datatype D is a (strong) monad. The unit η is the constructor now while the multiplication μ is "concatenation" of laters:

$$\begin{aligned}
&\mu : \mathsf{D}\,(\mathsf{D}\,X) \to \mathsf{D}\,X \\
&\mu\,(\mathsf{now}\, c) = c \\
&\mu\,(\mathsf{later}\, c) = \mathsf{later}\,(\mu\, c)
\end{aligned}$$

We denote by $\mathsf{bind} : (X \to \mathsf{D}\,Y) \to \mathsf{D}\,X \to \mathsf{D}\,Y$ the bind operation and by $\mathsf{str} : X \times \mathsf{D}\,Y \to \mathsf{D}\,(X \times Y)$ the strength operation of D (which it has uniquely, as any monad on **Set**).

Proposition 3.1. *The delay datatype* D *is a monad.*

We already noted that not all quotients are effective. But the quotient we are interested in, namely $\mathsf{D}\,X/{\approx}$ for a type X, is indeed effective. Notice that, by Proposition 2.2, it suffices to prove $\|c_1 \approx c_2\| \to c_1 \approx c_2$ for all $c_1, c_2 : \mathsf{D}\,X$.

Lemma 3.1. *For all $c_1, c_2 : \mathsf{D}\,X$, there exists a constant endofunction on $c_1 \approx c_2$. Therefore, the type $\|c_1 \approx c_2\| \to c_1 \approx c_2$ is inhabited.*

Proof. Let $c_1, c_2 : \mathsf{D}\,X$. We consider the following function.

$$\begin{aligned}
&\mathsf{canon}{\approx} : c_1 \approx c_2 \to c_1 \approx c_2\\
&\mathsf{canon}{\approx}\,(\downarrow_{\approx} \mathsf{now}_{\downarrow}\, p_2) = \downarrow_{\approx} \mathsf{now}_{\downarrow}\, p_2\\
&\mathsf{canon}{\approx}\,(\downarrow_{\approx} (\mathsf{later}_{\downarrow}\, p_1)\, \mathsf{now}_{\downarrow} = \downarrow_{\approx} (\mathsf{later}_{\downarrow}\, p_1)\, \mathsf{now}_{\downarrow}\\
&\mathsf{canon}{\approx}\,(\downarrow_{\approx} (\mathsf{later}_{\downarrow}\, p_1)\, (\mathsf{later}_{\downarrow}\, p_2)) = \mathsf{later}_{\approx}\, (\mathsf{canon}{\approx}\,(\downarrow_{\approx} p_1\, p_2))\\
&\mathsf{canon}{\approx}\,(\mathsf{later}_{\approx}\, p) = \mathsf{later}_{\approx}\, (\mathsf{canon}{\approx}\, p)
\end{aligned}$$

The function $\mathsf{canon}{\approx}$ canonizes a given weak bisimilarity proof by maximizing the number of applications of the constructor $\mathsf{later}_{\approx}$. This function is indeed constant, i.e., one can prove $\prod_{p_1,p_2:c_1\approx c_2} p_1 \cong p_2$ for all $c_1, c_2 : \mathsf{D}\,X$, where the relation $\cong$ is strong bisimilarity on proofs of $c_1 \approx c_2$, coinductively defined by the rules:

$$\frac{}{\downarrow_{\approx} p_1\, p_2 \cong \downarrow_{\approx} p_1\, p_2} \qquad \frac{p_1 \cong p_2}{\mathsf{later}_{\approx}\, p_1 \cong \mathsf{later}_{\approx}\, p_2}$$

Similarly to extensionality of delayed computations, we assume that strongly bisimilar weak bisimilarity proofs are equal, i.e., that we have an inhabitant for

$$\approx\mathsf{Ext} = \prod_{\{X:\mathcal{U}\}} \prod_{\{c_1,c_2:\mathsf{D}\,X\}} \prod_{p_1,p_2:c_1\approx c_2} p_1 \cong p_2 \to p_1 \equiv p_2$$

□

Theorem 3.1. *The quotient $\mathsf{D}\,X/{\approx}$ is effective.*

Notice that the above theorem is trivial when working with the equivalent relation $\approx'$ instead of $\approx$. This is because the relation $\approx'$ is propositional and quotients by propositional relations are always effective.

3.2 Quotiented delay datatype

Remember that Martin-Löf type theory does not have quotient types. We discussed in Section 2.2 two possible ways to overcome this issue: using setoids or extending the theory with inductive-like quotients à la Hofmann. In the first approach to quotients, the quotiented delay datatype is a monad, while in the second approach we need to employ an additional choice principle among those described in Section 2.3.

In the quotients-as-setoids approach, it is trivial to define the corresponding (strong) monad structure on the quotient of D by $\approx$. The role of the quotiented datatype is played by the setoid functor $\hat{\mathsf{D}}$, defined by

$\hat{\mathsf{D}}\,(X, R) = (\mathsf{D}\,X, \approx_R)$. The unit $\hat{\eta}$ and multiplication $\hat{\mu}$ are just η and μ together with proofs of that the appropriate equivalences are preserved. The unit $\hat{\eta}$ is a setoid morphism from (X, R) to $(\mathsf{D}\,X, \approx_R)$, as $x_1\,R\,x_2 \to \mathsf{now}\,x_1 \approx_R \mathsf{now}\,x_2$ by definition of $\approx_R$. The multiplication $\hat{\mu}$ is a setoid morphism from $(\mathsf{D}\,(\mathsf{D}\,X), \approx_{\approx_R})$ to $(\mathsf{D}\,X, \approx_R)$, since $c_1 \approx_{\approx_R} c_2 \to \mu\,c_1 \approx_R \mu\,c_2$ for all $c_1, c_2 : \mathsf{D}\,(\mathsf{D}\,X)$. The monad laws hold up to $\approx_R$, since they hold up to $\sim_R$.

In the quotients-as-sets approach, we define $\mathsf{D}_\approx X = \mathsf{D}\,X/\!\approx$. Let us try to equip the functor $\mathsf{D}_\approx$ with a monad structure. Let X be a type. As the unit $\eta_\approx : X \to \mathsf{D}_\approx X$, we can take $\eta_\approx = [\_] \circ \mathsf{now}$. But when we try to construct a multiplication $\mu_\approx : \mathsf{D}_\approx (\mathsf{D}_\approx X) \to \mathsf{D}_\approx X$, we get stuck immediately. Indeed, $\mu_\approx$ must be of the form $\mathsf{lift}\,\mu'_\approx\,p$ for some $\mu'_\approx : \mathsf{D}\,(\mathsf{D}_\approx X) \to \mathsf{D}_\approx X$ with $p : \mathsf{compat}\,\mu'_\approx$, but we cannot define such $\mu'_\approx$ and p. The problem lies in the coinductive nature of the delay datatype. We are trying to construct a function of type $\mathsf{D}\,(\mathsf{D}_\approx X) \to \mathsf{D}_\approx X$ that sends a converging computation to its converging value and a non-terminating one to the equivalence class of non-termination. This discontinuity makes constructing such a function problematic. Moreover, one can show that a right inverse of $[\_] : \mathsf{D}\,X \to \mathsf{D}_\approx X$, i.e., a canonical choice of representative for each equivalence class in $\mathsf{D}_\approx$, is not definable [75, Ch. 5.4.3]. Therefore, we cannot even construct $\mu'_\approx$ as a composition $[\_] \circ \mu''_\approx$ with $\mu''_\approx : \mathsf{D}\,(\mathsf{D}_\approx X) \to \mathsf{D}\,X$, since we do not know how to define $\mu''_\approx(\mathsf{now}\,q)$ for $q : \mathsf{D}\,X/\!\approx$.

A function $\mu'_\approx$ would be constructable, if the type $\mathsf{D}\,(\mathsf{D}_\approx X)$ were a quotient of $\mathsf{D}\,(\mathsf{D}\,X)$ by the equivalence relation $\mathsf{D}\!\approx$ (remember that $\mathsf{D}\!\approx$ is a synonym of $\sim_\approx$, the functorial lifting of $\approx$ from $\mathsf{D}\,X$ to $\mathsf{D}\,(\mathsf{D}\,X)$). In fact, the function $[\_] \circ \mu : \mathsf{D}\,(\mathsf{D}\,X) \to \mathsf{D}_\approx X$ is $\mathsf{D}\!\approx$-compatible, since $x_1(\mathsf{D}\!\approx)x_2 \to \mu\,x_1 \approx \mu\,x_2$, and therefore the elimination principle would do the job. But how "different" are $\mathsf{D}\,(\mathsf{D}_\approx X)$ and the quotient $\mathsf{D}\,(\mathsf{D}\,X)/\mathsf{D}\!\approx$? More generally, how "different" are $\mathsf{D}\,(X/R)$ and the quotient $\mathsf{D}\,X/\mathsf{D}\,R$, for a given type X and equivalence relation R on X?

3.2.1 A solution using LPO

Intuitively, the inhabitants of $\mathsf{D}_\approx X$ are the elements of X and the non-terminating computation never. But notice that the type $\mathsf{D}_\approx X$ is not isomorphic to $\mathsf{Maybe}\,X = X + 1$ constructively. One can always construct the map:

$$\begin{aligned}
&\mathsf{Maybe2D}_\approx : \mathsf{Maybe}\,X \to \mathsf{D}_\approx X\\
&\mathsf{Maybe2D}_\approx\,(\mathsf{inl}\,x) = [\mathsf{now}\,x]\\
&\mathsf{Maybe2D}_\approx\,(\mathsf{inr}\,*) = [\mathsf{never}]
\end{aligned}$$

But $\mathsf{D}_{\approx}X \cong \mathsf{Maybe}\,X$ is equivalent to the *limited principle of omniscience* (LPO), an instance of the principle of excluded middle.

$$\mathsf{LPO} = \prod_{f:\mathbb{N}\to 2} \left(\sum_{n:\mathbb{N}} f\,n \equiv 1\right) + \left(\prod_{n:\mathbb{N}} f\,n \equiv 0\right)$$

Proposition 3.2. *The isomorphism* $\mathsf{D}_{\approx}X \cong \mathsf{Maybe}\,X$ *is logically equivalent to* LPO.

Proof. Assuming LPO, one can decide if a computation $c : \mathsf{D}\,X$ is terminating or not. In fact, construct a stream $f_c : \mathbb{N} \to 2$ that is everywhere 0 expect for the unique position $n : \mathbb{N}$ such that $c = \mathsf{later}^n\,(\mathsf{now}\,x)$ (if such an n exists), in which $f_c\,n = 1$. Using LPO we can decide if f_c is everywhere 0, in which case $c = \mathsf{never}$, or if there exists a position n for which $f_c\,n = 1$, in which case $c = \mathsf{later}^n\,(\mathsf{now}\,x)$ for some $x : X$. Clearly, if we can decide if a computation is terminating, the types $\mathsf{D}_{\approx}\,X$ and $\mathsf{Maybe}\,X$ are isomorphic.

Vice versa, assume $\mathsf{D}_{\approx}\,X \cong \mathsf{Maybe}\,X$. In particular, there exists a function $\mathsf{D}_{\approx}\mathsf{2Maybe} : \mathsf{D}_{\approx}\,X \to \mathsf{Maybe}\,X$ inverse of $\mathsf{Maybe2D}_{\approx}$. Given a binary stream $f : \mathbb{N} \to 2$, construct the computation $c_f : \mathsf{D}\,1$ that has as many laters as the length of the maximal prefix of f containing only the element 0. If $\mathsf{D}_{\approx}\mathsf{2Maybe}\,[c_f] = \mathsf{inl}\,*$, then $[c_f] = \mathsf{Maybe2D}_{\approx}\,(\mathsf{D}_{\approx}\mathsf{2Maybe}\,[c_f]) = \mathsf{Maybe2D}_{\approx}\,(\mathsf{inl}\,*) = [\mathsf{now}\,*]$. Since, by Theorem 3.1, the quotient $\mathsf{D}_{\approx}\,X$ is effective, we obtain $c_f \approx \mathsf{now}\,*$, i.e. $c_f = \mathsf{later}^n\,(\mathsf{now}\,*)$ for a certain $n : \mathbb{N}$. The latter natural number corresponds to the position n in which $f\,n \equiv 1$. Instead, if $\mathsf{D}_{\approx}\mathsf{2Maybe}\,[c_f] = \mathsf{inr}\,*$, then $[c_f] = \mathsf{Maybe2D}_{\approx}\,(\mathsf{D}_{\approx}\mathsf{2Maybe}\,[c_f]) = \mathsf{Maybe2D}_{\approx}\,(\mathsf{inr}\,*) = [\mathsf{never}]$, and therefore $c_f \approx \mathsf{never}$. This implies that $f\,n \equiv 0$ for all $n : \mathbb{N}$. □

Since Maybe is a monad, assuming LPO we have that the functor $\mathsf{D}_{\approx}$ has a monad structure as well.

3.2.2 A solution using weak countable choice

In previous work [30], we proved that the quotiented delay datatype is a monad assuming the axiom of countable choice. Here we show that weak countable choice, introduced in Section 2.3.3, is enough to construct the monad structure.

In order to use the results of Section 2.3.3, we think of possibly non-terminating computations as streams. More precisely, let X be a type and $c : \mathsf{D}\,X$. Now c can be thought of as a stream $\varepsilon\,c : \mathbb{N} \to X + 1$ with at most

one value element in the left summand X.

$$\begin{aligned}
&\varepsilon : \mathsf{D}\, X \to (\mathbb{N} \to X + 1)\\
&\varepsilon\, (\mathsf{now}\, x)\, \mathsf{zero} = \mathsf{inl}\, x\\
&\varepsilon\, (\mathsf{later}\, c)\, \mathsf{zero} = \mathsf{inr}\, \star\\
&\varepsilon\, (\mathsf{now}\, x)\, (\mathsf{suc}\, n) = \mathsf{inr}\, \star\\
&\varepsilon\, (\mathsf{later}\, c)\, (\mathsf{suc}\, n) = \varepsilon\, c\, n
\end{aligned}$$

Conversely, from a stream $f : \mathbb{N} \to X + 1$, one can construct a computation $\pi\, f : \mathsf{D}\, X$. This computation corresponds to the "truncation" of the stream to its first value in X.

$$\begin{aligned}
&\pi : (\mathbb{N} \to X + 1) \to \mathsf{D}\, X\\
&\pi\, f = \mathsf{case}\ f\, \mathsf{zero}\ \mathsf{of}\\
&\qquad \mathsf{inl}\, x \mapsto \mathsf{now}\, x\\
&\qquad \mathsf{inr}\, \star \mapsto \mathsf{later}\, (\pi\, (f \circ \mathsf{suc}))
\end{aligned}$$

We see that $\mathsf{D}\, X$ is isomorphic to a subset of $\mathbb{N} \to X + 1$ in the sense that, for all $c : \mathsf{D}\, X$, $\pi\, (\varepsilon\, c) \sim c$, and therefore $\pi(\varepsilon\, c) \equiv c$ by delayed computation extensionality. More precisely, we have the following isomorphism:

$$\mathsf{D}\, X \cong \sum_{f : \mathbb{N} \to X + 1} \mathsf{isProp} \left(\sum_{n : \mathbb{N}} \sum_{x : \mathbb{X}} f\, n \equiv \mathsf{inl}\, x \right) \tag{3.1}$$

As already pointed out before, if one could prove that the type $\mathsf{D}\, (\mathsf{D}_{\approx}\, X)$ is the carrier of a quotient of $\mathsf{D}\, (\mathsf{D}\, X)$ by the equivalence relation $\mathsf{D} \approx$, then one can define multiplication. By Proposition 2.4, we know that if there exists a surjective map $b : \mathsf{D}\, (\mathsf{D}\, X) \to \mathsf{D}\, (\mathsf{D}_{\approx}\, X)$, then the type $\mathsf{D}\, (\mathsf{D}_{\approx}\, X)$ is the carrier of a quotient of $\mathsf{D}\, (\mathsf{D}\, X)$ by the equivalence relation $\equiv_b$. We have a map of type $\mathsf{D}\, (\mathsf{D}\, X) \to \mathsf{D}\, (\mathsf{D}_{\approx}\, X)$, namely $\mathsf{D}[\_]$. We show that the map $\mathsf{D}[\_]$ is surjective under the assumption of the principle of weak countable choice.

Theorem 3.2. *Assume* $\mathsf{wac}\omega : \mathsf{WAC}\omega$. *Let* X *be a type and* R *an equivalence relation on* X. *The function* $\mathsf{D}[\_] : \mathsf{D}\, X \to \mathsf{D}\, (X/R)$ *is surjective.*

Proof. Let $c : \mathsf{D}\, (X/R)$. We need to prove $\| \sum_{d : \mathsf{D}\, X} \mathsf{D}[\_]\, d \equiv c \|$. The proof goes via the axiom of weak countable choice. Therefore we construct a predicate $P : \mathbb{N} \to \mathcal{U}$ together with a proof $ip : \prod_{m<n} \mathsf{isProp}\, (P\, m) + \mathsf{isProp}\, (P\, n)$ and a function $p : \prod_{n : \mathbb{N}} \| P\, n \|$.

Let $P\, n = \sum_{x : X+1} ([\_] + \mathsf{id})\, x \equiv \varepsilon\, c\, n$. We give a proof $p : \prod_{n : \mathbb{N}} \| P\, n \|$. Let $n : \mathbb{N}$, we proceed by case analysis on $\varepsilon\, c\, n : X/R + 1$:

- if $\varepsilon\, c\, n = \mathsf{inl}\, q$ for some equivalence class $q : X/R$, then by surjectivity of $[\_]$, we have $\| \sum_{x : X} [x] \equiv q \|$, and $\| P\, n \|$ follows using the elimination principle of propositional truncation;

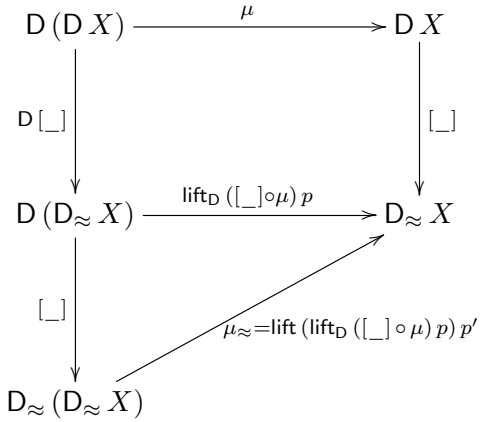
- if $\varepsilon\, c\, n = \mathsf{inr}\, *$, we return $|\mathsf{inr}\, *, \mathsf{refl}| : \|P\, n\|$.

We now give a proof $ip : \prod_{m<n} \mathsf{isProp}\,(P\, m) + \mathsf{isProp}\,(P\, n)$. Suppose $m < n$, we proceed by case analysis on $\varepsilon\, c\, m$ and on $\varepsilon\, c\, n$:

- if $\varepsilon\, c\, m = \mathsf{inl}\, q_1$ and $\varepsilon\, c\, n = \mathsf{inl}\, q_2$, for some equivalence classes $q_1, q_2 : X/R$, then we obtain a contradiction, since there is at most one entry from X/R in the stream $\varepsilon\, c$;
- if $\varepsilon\, c\, m = \mathsf{inr}\, *$, we are done since $P\, m$ is a proposition;
- if $\varepsilon\, c\, n = \mathsf{inr}\, *$, we are done since $P\, n$ is a proposition.

We can apply $\mathsf{WAC}\omega$ to the predicate P and the terms ip and p. We obtain a proof of $\left\|\prod_{n:\mathbb{N}} \sum_{x:X+1} ([\_] + \mathsf{id})\, x \equiv \varepsilon\, c\, n\right\|$. Using the elimination principle of propositional truncation, we assume that there exists a stream $g : \mathbb{N} \to X + 1$ such that $([\_] + \mathsf{id})\,(g\, n) \equiv \varepsilon\, c\, n$, for all $n : \mathbb{N}$, and we need to construct a computation $d : \mathsf{D}\, X$ such that $\mathsf{D}[\_]\, d \equiv c$. We define $d = \pi\, g$. It is not difficult to see that $\mathsf{D}[\_]\,(\pi\, g) \equiv \pi\,((([\_] + \mathsf{id}) \circ g)$, and by hypothesis we have $([\_] + \mathsf{id}) \circ g \equiv \varepsilon\, c$. We are done, since $c \equiv \pi\,(\varepsilon\, c)$. □

Theorem 3.2 and Proposition 2.4 tell us that, under the assumption of $\mathsf{WAC}\omega$, the type $\mathsf{D}\,(\mathsf{D}_{\approx}\, X)$ is the carrier of a quotient of $\mathsf{D}\,(\mathsf{D}\, X)$ by the equivalence relation $\equiv_{\mathsf{D}[\_]}$. One can then construct multiplication $\mu_{\approx} : \mathsf{D}_{\approx}\,(\mathsf{D}_{\approx}\, X) \to \mathsf{D}_{\approx}\, X$ in the following way:



In the diagram we wrote $\mathsf{lift}_{\mathsf{D}}$ for the dependent eliminator of $\mathsf{D}\,(\mathsf{D}_{\approx}\, X)$. The above diagram makes sense only if we can construct the two compatibility proofs p and p'.

- We have to prove that $\mathsf{D}[\_]\, c \equiv \mathsf{D}[\_]\, d$ implies $[\mu\, c] \equiv [\mu\, d]$, for all $c, d : \mathsf{D}\,(\mathsf{D}\, X)$. Using the soundness property of quotients, it is enough to show $\mu\, c \approx \mu\, d$. First, notice that $\mathsf{D}[\_]\, c \equiv \mathsf{D}[\_]\, d$ implies $\mathsf{D}[\_]\, c \sim$

$\mathsf{D}[\_]\,d$, which in turns implies $c \sim_\approx d$, since by Theorem 3.1 the quotient $\mathsf{D}_\approx X$ is effective. A easy corecursive construction shows that $c \sim_\approx d$ implies $\mu\, c \approx \mu\, d$.

- We have to prove that $c \approx d$ implies $\mathsf{lift}_\mathsf{D}\,([\_] \circ \mu)\, p\, c \equiv \mathsf{lift}_\mathsf{D}\,([\_] \circ \mu)\, p\, d$, for all $c, d : \mathsf{D}\,(\mathsf{D}_\approx X)$. Using the elimination principle of quotients, we have to prove that $\mathsf{D}[\_]\,c \approx \mathsf{D}[\_]\,d$ implies $\mathsf{lift}_\mathsf{D}\,([\_] \circ \mu)\, p\,(\mathsf{D}[\_]\,c) \equiv \mathsf{lift}_\mathsf{D}\,([\_] \circ \mu)\, p\,(\mathsf{D}[\_]\,d)$, for all $c, d : \mathsf{D}\,(\mathsf{D}\,X)$. Using the computation rule of lift_D and the soundness principle of quotients, we are left to prove $\mu\, c \approx \mu\, d$. Notice that $\mathsf{D}[\_]\,c \approx \mathsf{D}[\_]\,d$ implies $c \approx_\approx d$, again by effectiveness of $\mathsf{D}_\approx X$. An easy corecursive construction shows that $c \approx_\approx d$ implies $\mu\, c \approx \mu\, d$.

We omit the proof of the monad laws, which is the easy part—essentially the proofs for the unquotiented delay datatype carry over.

Theorem 3.3. *Assuming* $\mathsf{WAC}\omega$*, the type functor* $\mathsf{D}_\approx$ *is a monad.*

3.2.3 A solution using propositional choice

In this section, we show how to construct a monad structure on the quotiented delay datatype using the axiom of propositional choice introduced in Section 2.3.4. This solution was suggested to us by Martin Escardó. In a recent paper [46], Escardó and Knapp describe an implementation of the partial map classifier associated to Rosolini's dominance (that we discuss in Sections 3.5 and 4.5.1). Their datatype carries a monad structure assuming a form of propositional choice. More precisely, they assume propositional choice from Rosolini propositions. A proposition X being Rosolini is equivalent to the type $\|\sum_{c:\mathsf{D}\,1} (c\downarrow * \leftrightarrow X)\|$ being inhabited. They conjecture their monad to be isomorphic to the quotiented delay monad.

First, we show that the functor $\mathsf{D}_\approx$ is specified by a container [3].

Proposition 3.3. *The following isomorphism holds:*

$$\mathsf{D}_\approx X \cong \sum_{x:\mathsf{D}_\approx 1} (x \equiv [\mathsf{now}\, *] \to X)$$

Proof. We define a $\approx$-compatible map $f : \mathsf{D}\,X \to \sum_{x:\mathsf{D}_\approx 1} (x \equiv [\mathsf{now}\, *] \to X)$.

$$f : \mathsf{D}\,X \to \sum_{x:\mathsf{D}_\approx 1} (x \equiv [\mathsf{now}\, *] \to X)$$
$$f\, c = ([\mathsf{D}!\, c], p)$$

where $!$ is the unique map into 1 and $p : [\mathsf{D}\,!\,c] \equiv [\mathsf{now}\, *] \to X$ is constructed as follows. Assume $[\mathsf{D}\,!\,c] \equiv [\mathsf{now}\, *]$, which is equivalent to $\mathsf{D}\,!\,c \downarrow *$. By an

easy inductive proof, the latter implies that there exists $x : X$ such that $c{\downarrow}x$. The output of p is exactly x. The function f is clearly $\approx$-compatible: if $c_1 \approx c_2$ then $\mathsf{D}\,!\,c_1 \approx \mathsf{D}\,!\,c_2$; moreover, if both $\mathsf{D}\,!\,c_1$ and $\mathsf{D}\,!\,c_2$ converge to $*$, we have that $c_1{\downarrow}x_1$ and $c_2{\downarrow}x_2$ for some $x_1, x_2 : X$, but $x_1 \equiv x_2$ since c_1 and c_2 are weakly bisimilar. Therefore, using the elimination principle of quotient types, we obtain a function $f_{\approx} : \mathsf{D}_{\approx}X \to \sum_{x:\mathsf{D}_{\approx}1} (x \equiv [\mathsf{now}\,*] \to X)$.

Vice versa, we define a map $g : \prod_{c:\mathsf{D}\,1} (c{\downarrow}* \to X) \to \mathsf{D}\,X$ by corecursion.

$$\begin{aligned} g &: \prod_{c:\mathsf{D}\,1} (c{\downarrow}* \to X) \to \mathsf{D}\,X \\ g\,(\mathsf{now}\,*)\,h &= \mathsf{now}\,(h\,(\mathsf{now}_{\downarrow}\,\mathsf{refl})) \\ g\,(\mathsf{later}\,c)\,h &= \mathsf{later}\,(g\,c\,(h \circ \mathsf{later}_{\downarrow})) \end{aligned}$$

Using the elimination principle of quotient types, it is not difficult to lift the map g to a map $g_{\approx} : \sum_{x:\mathsf{D}_{\approx}1} (x \equiv [\mathsf{now}\,*] \to X) \to \mathsf{D}_{\approx}X$. The maps $f_{\approx}$ and $g_{\approx}$ are each other inverses. □

Assuming the propositional axiom of choice, we are able to define a monad structure on the quotiented delay datatype.

Theorem 3.4. *Assuming* PAC, $\mathsf{D}_{\approx}$ *is a monad.*

Proof. In Proposition 3.3 we showed that $\mathsf{D}_{\approx}$ is (isomorphic to) a functor specified by a container [3]: the set of shapes is $S = \mathsf{D}_{\approx}1$, while the set of positions is $P\,x = x \equiv [\mathsf{now}\,*]$. Therefore, $\mathsf{D}_{\approx}$ carries a monad structure if and only if it comes with certain extra structure [8], namely

$$\mathsf{e} : \mathsf{D}_{\approx}1 \qquad \bullet : \prod_{x:\mathsf{D}_{\approx}1} (x \equiv [\mathsf{now}\,*] \to \mathsf{D}_{\approx}1) \to \mathsf{D}_{\approx}1$$

$$q_0 : \prod_{x:\mathsf{D}_{\approx}1} \prod_{v:x\equiv[\mathsf{now}\,*]\to\mathsf{D}_{\approx}1} x \bullet v \equiv [\mathsf{now}\,*] \to x \equiv [\mathsf{now}\,*]$$

$$q_1 : \prod_{x:\mathsf{D}_{\approx}1} \prod_{v:x\equiv[\mathsf{now}\,*]\to\mathsf{D}_{\approx}1} \prod_{p:x\bullet v\equiv[\mathsf{now}\,*]} v\,(q_0\,x\,v\,p) \equiv [\mathsf{now}\,*]$$

satisfying the equations

$$x \bullet (\lambda\_.\,\mathsf{e}) \equiv x \qquad \mathsf{e} \bullet (\lambda\_.\,x) \equiv x$$

$$(x \bullet v) \bullet (\lambda p.\,w\,(q_0\,x\,v\,p)\,(q_1\,x\,v\,p)) \equiv x \bullet (\lambda p.\,v\,p \bullet w\,p)$$

Notice that, in general, more equalities between positions are required to hold. In our case, these equations are all trivial, since the type $x \equiv [\mathsf{now}\,*]$ is a proposition, for all $x : \mathsf{D}_{\approx}1$.

We take $\mathsf{e} = [\mathsf{now}\,*]$ and we define $\bullet$ in two steps. First we construct a map $\bullet_{\mathsf{D}} : \prod_{c:\mathsf{D}\,1} (c{\downarrow} * \to \mathsf{D}\,1) \to \mathsf{D}\,1$, similar to the map g introduced in the proof of Proposition 3.3.

$$\begin{aligned}
&\bullet_{\mathsf{D}} : \prod_{c:\mathsf{D}\,1} (c{\downarrow} * \to \mathsf{D}\,1) \to \mathsf{D}\,1\\
&\mathsf{now}\,* \bullet_{\mathsf{D}} h = h\,(\mathsf{now}_{\downarrow}\,\mathsf{refl})\\
&\mathsf{later}\,c \bullet_{\mathsf{D}} h = \mathsf{later}\,(c \bullet_{\mathsf{D}} (h \circ \mathsf{later}_{\downarrow}))
\end{aligned}$$

Using the elimination principle of quotient types twice, we can lift $\bullet_{\mathsf{D}}$ to a function $\bullet' : \prod_{x:\mathsf{D}_{\approx}1} (x \equiv [\mathsf{now}\,*] \to \mathsf{D}\,1)/(x \equiv [\mathsf{now}\,*] \to \approx) \to \mathsf{D}_{\approx}1$, where as usual the equivalence relation $x \equiv [\mathsf{now}\,*] \to \approx$ is the lifting of $\approx$ to the function space $x \equiv [\mathsf{now}\,*] \to \mathsf{D}\,1$. By propositional choice, since $x \equiv [\mathsf{now}\,*]$ is a proposition, we have that the types $(x \equiv [\mathsf{now}\,*] \to \mathsf{D}\,1)/(x \equiv [\mathsf{now}\,*] \to\approx)$ and $x \equiv [\mathsf{now}\,*] \to \mathsf{D}_{\approx}\,1$ are isomorphic. Applying this isomorphism to $\bullet'$, we are able to define $\bullet$.

The terms q_0 and q_1, together with the proofs of the three equations, are constructed using the elimination principle of quotient types. □

Notice that in the above theorem we employed only a restricted form of the axiom of propositional choice. More precisely, we employed what Escardó and Knapp call propositional choice from Rosolini propositions [46]. In fact, it is not difficult to prove that the proposition $x \equiv [\mathsf{now}\,*]$ is Rosolini, since the proposition $\| \sum_{c:\mathsf{D}\,1} (c{\downarrow} * \leftrightarrow x \equiv [\mathsf{now}\,*]) \|$ holds.

3.3 A monad or an arrow?

Hughes [55] has proposed arrows as a generalization of monads. Jacobs et al. [59] have sorted out their mathematical theory.

We have seen that it takes a semi-classical principle to show that quotienting the functor D by weak bisimilarity preserves its monad structure. In contrast, quotienting the corresponding profunctor KD, defined by $\mathsf{KD}\,X\,Y = X \to \mathsf{D}\,Y$, by pointwise weak bisimilarity can easily be shown to preserve its (strong) arrow structure (whose Freyd category is isomorphic to the Kleisli category of the monad) without invoking such principles.

Indeed, the arrow structure on KD is given by $\mathsf{pure} : (X \to Y) \to \mathsf{KD}\,X\,Y$, $\mathsf{pure}\,f = \eta \circ f$ and $\lll\colon \mathsf{KD}\,Y\,Z \to \mathsf{KD}\,X\,Y \to \mathsf{KD}\,X\,Z$, $\ell \lll k = \mathsf{bind}\,\ell \circ k$.

Now, define the quotiented profunctor by $\overline{\mathsf{KD}}\,X\,Y = (X \to \mathsf{D}\,Y)/(X \to \approx)$. We can define $\overline{\mathsf{pure}} : (X \to Y) \to \overline{\mathsf{KD}}\,X\,Y$ straightforwardly by $\overline{\mathsf{pure}}\,f = [\mathsf{pure} f]$. But we can also construct $\overline{\lll} : \overline{\mathsf{KD}}\,Y\,Z \to \overline{\mathsf{KD}}\,X\,Y \to \overline{\mathsf{KD}}\,X\,Z$ as $\ell \,\overline{\lll}\, k = \mathsf{lift}_2\,(\lll)\,p\,\ell\,k$, where p is an easy proof of $\ell_1\ (Y \to \approx)\ \ell_2 \to k_1\ (X \to \approx)\ k_2 \to (\ell_1 \lll k_1)\ (X \to \approx)\ (\ell_2 \lll k_2)$.

This works entirely painlessly, as there is no need in this construction for a coercion $(X \to Y/\!\approx) \to (X \to Y)/(X \to \approx)$. From the beginning, we quotient the relevant function types here rather than their codomains.

There are some further indications that quotienting the arrow may be a sensible alternative to quotienting the monad. In particular, the work by Cockett et al. [34] suggests that working with finer quotients of the arrow considered here may yield a setting for dealing with computational complexity rather computability constructively.

3.4 Quotiented delay delivers free ωcppos

In this section, we show that the type $\mathsf{D}_{\approx} X$ is the free ω-complete pointed partial order over X.

3.4.1 Free ωcppo structure up to $\approx$

First, we show that the type $\mathsf{D}\, X$ is endowed with a ωcppo structure up to $\approx$. Moreover, it is the free ωcppo up to $\approx$ over X. The construction performed in this subsection will be lifted to the quotient $\mathsf{D}\, X/\!\approx$ in Section 3.4.2. Following [26], we introduce an information order on $\mathsf{D}\, X$:

$$\frac{c_1 \downarrow x \quad c_2 \downarrow x}{c_1 \sqsubseteq c_2} \qquad \frac{c_1 \sqsubseteq c_2}{\mathsf{later}\, c_1 \sqsubseteq \mathsf{later}\, c_2} \qquad \frac{c_1 \sqsubseteq c_2}{\mathsf{later}\, c_1 \sqsubseteq c_2}$$

The type $c_1 \sqsubseteq c_2$ is inhabited when $c_1 \approx c_2$, but also when c_1 has some (possibly infinitely many) laters more than c_2. The relation $\sqsubseteq$ is reflexive and transitive. Moreover, it is antisymmetric up to $\approx$, i.e., $c_1 \sqsubseteq c_2 \to c_2 \sqsubseteq c_1 \to c_1 \approx c_2$, for all $c_1, c_2 : \mathsf{D}\, X$. Notice also that the relation $\sqsubseteq$ is not propositional. The least element is the non-terminating computation never.

We define a binary operation race on $\mathsf{D}\, X$ that returns the computation with the least number of laters. If two computations c_1 and c_2 converge simultaneously, $\mathsf{race}\, c_1\, c_2$ returns c_1.

$$\begin{aligned}
&\mathsf{race} : \mathsf{D}\, X \to \mathsf{D}\, X \to \mathsf{D}\, X \\
&\mathsf{race}\, (\mathsf{now}\, x)\, c = \mathsf{now}\, x \\
&\mathsf{race}\, (\mathsf{later}\, c)\, (\mathsf{now}\, x) = \mathsf{now}\, x \\
&\mathsf{race}\, (\mathsf{later}\, c_1)\, (\mathsf{later}\, c_2) = \mathsf{later}\, (\mathsf{race}\, c_1\, c_2)
\end{aligned}$$

Notice that generally $\mathsf{race}\, c_1\, c_2$ is not an upper bound of c_1 and c_2, since the two computations may converge to different values. The binary operation race can be extended to an ω-operation $\omega\mathsf{race}$. The latter constructs the first converging element of a stream of computations. It is defined using the

auxiliary operation $\omega\mathsf{race}'$:

$$\begin{aligned}
&\omega\mathsf{race}' : (\mathbb{N} \to \mathsf{D}\,X) \to \mathbb{N} \to \mathsf{D}\,X \to \mathsf{D}\,X\\
&\omega\mathsf{race}'\; s\; n\; (\mathsf{now}\; x) = \mathsf{now}\; x\\
&\omega\mathsf{race}'\; s\; n\; (\mathsf{later}\; c) = \mathsf{later}\; (\omega\mathsf{race}'\; s\; (\mathsf{suc}\; n)\; (\mathsf{race}\; c\; (s\; n)))
\end{aligned}$$

The operation $\omega\mathsf{race}'$, when applied to a stream $s : \mathbb{N} \to \mathsf{D}\,X$, a number $n : \mathbb{N}$ and an computation $c : \mathsf{D}\,X$, constructs the first converging element of the stream $s' : \mathbb{N} \to \mathsf{D}\,X$, with $s'\;\mathsf{zero} = c$ and $s'\;(\mathsf{suc}\; k) = s\;(n + k)$. The operation $\omega\mathsf{race}$ is constructed by instantiating $\omega\mathsf{race}'$ with $n = \mathsf{zero}$ and $c = \mathsf{never}$. In this way, we have that the first converging element of s is the first converging element of s', since never diverges.

$$\begin{aligned}
&\omega\mathsf{race} : (\mathbb{N} \to \mathsf{D}\,X) \to \mathsf{D}\,X\\
&\omega\mathsf{race}\; s = \omega\mathsf{race}'\; s\; \mathsf{zero}\; \mathsf{never}
\end{aligned}$$

Generally, $\omega\mathsf{race}\, s$ is not an upper bound of s. But if the stream s is increasing, then $\omega\mathsf{race}\, s$ is the join of s, i.e., the following terms exist:

$$\omega\mathsf{raceisUB} : \prod_{s:\mathbb{N}\to\mathsf{D}\,X} \prod_{i:\mathsf{isIncr}\; s} \prod_{n:\mathbb{N}} s\; n \sqsubseteq \omega\mathsf{race}\; s$$

$$\omega\mathsf{raceisSupremum} : \prod_{s:\mathbb{N}\to\mathsf{D}\,X} \prod_{i:\mathsf{isIncr}\; s} \prod_{c:\mathsf{D}\,X} \left(\prod_{n:\mathbb{N}} s\; n \sqsubseteq c\right) \to \omega\mathsf{race}\; s \sqsubseteq c$$

where $\mathsf{isIncr}\; s$ states that the stream s is increasing wrt. $\sqsubseteq$, i.e. a chain.

So far we have showed that $(\mathsf{D}\,X, \sqsubseteq, \mathsf{never}, \omega\mathsf{race})$ is a ωcppo up to $\approx$. We prove that it is the free one over X. Let $(Y, \leq, \bot, \cup)$ be a ωcppo and $f : X \to Y$ a function. Every computation in $\mathsf{D}\,X$ defines an stream in Y.

$$\begin{aligned}
&\mathsf{cpt2chain}_f : \mathsf{D}\,X \to \mathbb{N} \to Y\\
&\mathsf{cpt2chain}_f(\mathsf{now}\; x)\; n = f\; x\\
&\mathsf{cpt2chain}_f(\mathsf{later}\; c)\; \mathsf{zero} = \bot\\
&\mathsf{cpt2chain}_f(\mathsf{later}\; c)\; (\mathsf{suc}\; n) = \mathsf{cpt2chain}_f\; c\; n
\end{aligned}$$

Given a computation $c = \mathsf{later}^n\; (\mathsf{now}\; x)$ (if $n = \omega$, then $c = \mathsf{never}$), the chain $\mathsf{cpt2chain}_f\; c$ looks as follows:

$$\underbrace{\bot \quad \bot \quad \ldots \quad \bot}_{n} \quad f\; x \quad f\; x \quad f\; x \quad \ldots$$

Since the latter is increasing wrt $\leq$, it is possible to extend the function f to a function $\widehat{f} : \mathsf{D}\,X \to Y$, $\widehat{f}\; c = \cup(\mathsf{cpt2chain}_f\; c)$. We have that $\widehat{f}\;(\mathsf{now}\; x) \equiv$

$f\,x$. Moreover $\widehat{f}$ is strict and continuous, i.e., the following terms exist:

$$\begin{aligned}
\mathsf{hatOrderpreserving} &: \prod_{c_1,c_2:\mathsf{D}\,X} c_1 \sqsubseteq c_2 \to \widehat{f}\,c_1 \leq \widehat{f}\,c_2 \\
\mathsf{hatStrict} &: \widehat{f}\,\mathsf{never} \leq \bot \\
\mathsf{hatContinuous} &: \prod_{s:\mathbb{N}\to\mathsf{D}\,X} \prod_{i:\mathsf{isIncr}\,s} \widehat{f}\,(\omega\mathsf{race}\,s) \leq \cup(\widehat{f}\circ s)
\end{aligned} \tag{3.2}$$

The last statement makes sense because, if s is increasing, then $\widehat{f}\circ s$ is also increasing, thanks to $\mathsf{hatOrderpreserving}$. Moreover, $\widehat{f}$ is $\approx$-compatible and it is the unique $\approx$-compatible map of the form $g : \mathsf{D}\,X \to Y$ that satisfies the inequalities in (3.2) and such that $g\,(\mathsf{now}\,x) \equiv f\,x$.

3.4.2 Lifting the construction to $\mathsf{D}_\approx X$

The first step we need to perform in order to lift all the constructions of Section 3.4.1 to $\mathsf{D}_\approx X$ is the lifting of the relation $\sqsubseteq$. Unfortunately, this cannot be done directly. In fact, if we try to define a binary relation $\sqsubseteq_\approx$ on $\mathsf{D}_\approx X$ as follows:

$$\begin{aligned}
&\sqsubseteq_\approx : \mathsf{D}_\approx X \to \mathsf{D}_\approx X \to \mathcal{U} \\
&\sqsubseteq_\approx = \mathsf{lift}_2\,\sqsubseteq\,p
\end{aligned}$$

we realize that we need to give a term p of type $\prod_{\{c_1,c_2,d_1,d_2:\mathsf{D}\,X\}} c_1 \approx d_1 \to c_2 \approx d_2 \to c_1 \sqsubseteq c_2 \equiv d_1 \sqsubseteq d_2$, which is not a true statement. In fact, let $c_1 = c_2 = \mathsf{now}\,x$ and $d_1 = d_2 = \mathsf{later}\,(\mathsf{now}\,x)$, then the type $c_1 \sqsubseteq c_2$ is a proposition, while the type $d_1 \sqsubseteq d_2$ is not.

In order to overcome this issue, we lift the propositional truncation of the relation $\sqsubseteq$ instead of the relation $\sqsubseteq$ directly, as follows:

$$\begin{aligned}
&\sqsubseteq_\approx : \mathsf{D}_\approx X \to \mathsf{D}_\approx X \to \mathcal{U} \\
&\sqsubseteq_\approx = \mathsf{lift}_2\,(\lambda\,c_1, c_2.\,\|c_1 \sqsubseteq c_2\|)\,p
\end{aligned}$$

where p is a proof of $\prod_{\{c_1,c_2,d_1,d_2:\mathsf{D}\,X\}} c_1 \approx d_1 \to c_2 \approx d_2 \to \|c_1 \sqsubseteq c_2\| \equiv \|d_1 \sqsubseteq d_2\|$, which can be proved with the help of proposition extensionality. The relation $\sqsubseteq_\approx$ is propositional and the proofs of reflexivity, transitivity and antisymmetry up to $\approx$ of the relation $\sqsubseteq$ lift straightforwardly to the relation $\sqsubseteq_\approx$.

Remark 3.2. There is an alternative way of lifting $\sqsubseteq$ to $\mathsf{D}_\approx X$. Following [17], we introduce a binary relation $\sqsubseteq'$ on $\mathsf{D}\,X$:

$$\frac{c \downarrow x}{\mathsf{now}\,x \sqsubseteq' c} \qquad \frac{c_1 \sqsubseteq' c_2}{\mathsf{later}\,c_1 \sqsubseteq' \mathsf{later}\,c_2} \qquad \frac{c \sqsubseteq' \mathsf{now}\,x}{\mathsf{later}\,c \sqsubseteq' \mathsf{now}\,x}$$

Notice the similarity with the definition of $\approx'$ in Remark 3.1. The relation $\sqsubseteq'$ is equivalent to $\sqsubseteq$, but it is propositional. This implies that $\sqsubseteq'$ is liftable to $\mathsf{D}_{\approx} X$:

$$\begin{aligned}\sqsubseteq'_{\approx} &: \mathsf{D}_{\approx} X \to \mathsf{D}_{\approx} X \to \mathcal{U}\\ \sqsubseteq'_{\approx} &= \mathsf{lift}_2 \sqsubseteq' p\end{aligned}$$

where p is a proof of $\prod_{\{c_1,c_2,d_1,d_2:\mathsf{D}\,X\}} c_1 \approx d_1 \to c_2 \approx d_2 \to c_1 \sqsubseteq' c_2 \equiv d_1 \sqsubseteq' d_2$, which is a true statement. We prefer to work with $\sqsubseteq$ instead of $\sqsubseteq'$ for the same reasons specified in Remark 3.1 about $\approx$ and $\approx'$.

In order to lift the operator $\omega\mathsf{race}$, we rely on the axiom of countable choice described in Section 2.3.2. Under the assumption of $\mathsf{AC}\omega$, we know from Theorem 2.2 that the map $\mathbb{N} \to [\_]$ is surjective. Therefore, by Proposition 2.4, the type $\mathbb{N} \to \mathsf{D}_{\approx} X$ is the carrier of a quotient of $\mathbb{N} \to \mathsf{D}\,X$ by the relation $\equiv_{\mathbb{N}\to[\_]}$. Notice that the relations $\equiv_{\mathbb{N}\to[\_]}$ and $\mathbb{N} \to \approx$ are equivalent, since $\mathsf{D}_{\approx} X$ is effective by Theorem 3.1. Therefore the type $\mathbb{N} \to \mathsf{D}_{\approx} X$ is also the carrier of a quotient of $\mathbb{N} \to \mathsf{D}\,X$ by the relation $\mathbb{N} \to \approx$. The map $\mathbb{N} \to [\_]$ sends streams to their equivalence classes. We write $\mathsf{lift}_{\mathbb{N}}$ for the dependent eliminator of $\mathbb{N} \to \mathsf{D}_{\approx} X$ considered as the quotient of $\mathbb{N} \to \mathsf{D}\,X$ by $\mathbb{N} \to \approx$. Using $\mathsf{lift}_{\mathbb{N}}$ we are able to lift $\omega\mathsf{race}$ to the quotient:

$$\begin{aligned}\omega\mathsf{race}_{\approx} &: \prod_{s:\mathbb{N}\to\mathsf{D}_{\approx} X} \mathsf{isIncr}\, s \to \mathsf{D}_{\approx} X\\ \omega\mathsf{race}_{\approx} &= \mathsf{lift}_{\mathbb{N}}\,(\lambda s, i.\,[\omega\mathsf{race}\, s])\, p\end{aligned}$$

where p is a proof of compatibility of the function $\lambda s, i.\,[\omega\mathsf{race}\, s]$ with the relation $\mathbb{N} \to \approx$, which is indeed the case. The proofs attesting that $(\mathsf{D}_{\approx} X, \sqsubseteq_{\approx}, [\mathsf{never}], \omega\mathsf{race}_{\approx})$ is the free ωcppo over X are obtained by directly lifting the corresponding proofs described in Section 3.4.1 to $\mathsf{D}_{\approx} X$, with the fundamental help of the elimination principle $\mathsf{lift}_{\mathbb{N}}$.

Given a map $f : X \to Y$ into a ωcppo Y, we also indicate with $\widehat{f} : \mathsf{D}_{\approx} X \to Y$ the unique ωcppo morphism extending f given by the universal property.

3.5 Partiality in homotopy type theory

The quotiented delay monad constitutes a possible way of representing partiality an an effect in type theory. Recently, Altenkirch et al. have constructed another datatype A for partiality in homotopy type theory [9]. Their construction makes use of higher inductive-inductive types and it resembles the implementation of Cauchy reals in the HoTT book [83, Ch. 11.3]. The datatype A delivers free ωcppos by construction and it carries a monad structure without recourse to choice principles. Higher

inductive inductive types rather than ordinary higher inductive types are needed because the join constructor $\cup$ takes as argument a proof that a given stream is increasing. So the type $\mathsf{A}\,X$ has to be introduced mutually with the partial order $\leq$ on it. Altenkirch et al. proved that $\mathsf{A}\,X$ is isomorphic to $\mathsf{D}\,X/{\approx}$ under the assumption of countable choice.

In this section we present yet another datatype for partiality in homotopy type theory, which does not make use of choice principles or higher inductive-inductive definitions. It is constructed using standard higher inductive types [83, Ch. 6.13]. As a consequence, our partiality datatype can be directly implemented in proof assistants such as Coq, which currently lack support for inductive-inductive types, and added to the HoTT library [16]. The datatype we present in this section is isomorphic to A and therefore, under the assumption of countable choice, also isomorphic to the quotiented delay datatype.

Our construction is based on the implementation of free countably-complete join semilattices as higher inductive types. A *countably-complete join semilattice* is a partially ordered set $(X, \leq)$ with a bottom element $\bot : X$ and a countable join operation $\bigvee : (\mathbb{N} \to X) \to X$. Notice that the join operation $\bigvee$ is defined for all streams, not just the increasing ones. A *countably-complete join semilattice morphism* between countably-complete join semilattices X and Y is a monotone function between X and Y which preserves bottom and joins.

Notice that countably-complete join semilattices admit an equational presentation as an infinitary algebraic theory. In homotopy type theory, it is possible to introduce the free object of an algebraic theory as a higher inductive type. This procedure is exemplified in the construction of the free group over a type [83, Ch. 6.11]. Let X be a type, the free countably-complete join semilattice on X is defined similarly as the following higher inductive type:

$$\frac{x : X}{\eta\, x : \mathcal{P}_\infty X} \qquad \frac{}{\bot : \mathcal{P}_\infty X} \qquad \frac{s : \mathbb{N} \to \mathcal{P}_\infty X}{\bigvee s : \mathcal{P}_\infty X}$$

$$\overline{x \vee y \equiv y \vee x} \qquad \overline{x \vee (y \vee z) \equiv (x \vee y) \vee z} \qquad \overline{x \vee x \equiv x} \qquad \overline{x \vee \bot \equiv x}$$

$$\overline{\textstyle\prod_{n:\mathbb{N}} s\, n \vee \bigvee s \equiv \bigvee s} \qquad \overline{\textstyle\bigvee s \vee x \equiv \bigvee(\lambda n.\, s\, n \vee x)}$$

the 0-truncation constructor

where the binary join operation is derived as $x \vee y = \bigvee(x, y, y, y, \dots)$. We define $x \leq y = x \vee y \equiv y$.

The type $\mathcal{P}_\infty X$ is the free countably-complete join semilattice on X by construction. In the types of its constructors, it is possible to identify the algebraic theory of countably-complete join semilattices. The 0-truncation constructor forces the type $\mathcal{P}_\infty X$ to be a set, i.e., to satisfy the principle of

uniqueness of identity proofs UIP. The dependent eliminator of $\mathcal{P}_\infty X$ is an induction principle expressing freeness.

It is a well-known fact that the free countably-complete join semilattice on a type X is the countable powerset of X, i.e., the type whose elements are the subsets of X with countable cardinality. The order $\leq$ is the inclusion order.

We define $\mathsf{S} = \mathcal{P}_\infty 1$. This type has $\top = \eta *$ as its top element, as we can prove by induction that $x \leq \top$ for all $x : \mathsf{S}$. The type S is the countable powerset of 1. It is important to realize that S is not isomorphic to Bool. $\bot$ corresponds to the empty subset of 1 and $\top$ corresponds to the full set 1. We can prove that $x \not\equiv \top$ implies $x \equiv \bot$ for all $x : \mathsf{S}$. But we cannot decide whether $x \equiv \bot$ or $x \equiv \top$. For a general $s : \mathbb{N} \to \mathsf{S}$, even if we happen to know that $s\,n$ is either $\bot$ or $\top$ for all $n : \mathbb{N}$, we cannot decide whether $s\,n \equiv \top$ for at least one $n : \mathbb{N}$, unless we assume LPO.

S happens to be also the initial σ-frame, i.e., a countably-complete join semilattice with finite meets which distribute over the joins. In fact, $\top$ is the top element and binary meets can be defined by induction.

S has an interesting relation with the free ωcppo on 1. If the latter exists, then they are isomorphic.

Proposition 3.4. *The free ωcppo on 1 is also the free countably-complete join semilattice on 1.*

Proof. Let $(X, \leq, \bot, \cup)$ be the free ωcppo on 1. We only need to construct a countable join operation $\bigvee$.

For any $x : X$, by the universal property of X, there exists a unique ω-continuous map $f_x : X \to \sum_{y:X} x \leq y$, since the latter is a ωcppo over 1. We can define a binary join operation as $x \vee y = \mathsf{fst}\,(f_x\,y)$. The countable join operation is defined as $\bigvee s = \cup s'$, where the stream s' is the majorization of s defined by induction as follows: $s'\,\mathsf{zero} = s\,\mathsf{zero}$ and $s'\,(\mathsf{suc}\,n) = s'\,n \vee s\,(\mathsf{suc}\,n)$. The stream s' is increasing and can be supplied as an argument of the join operation $\cup$ of X.

It is not difficult to show that the type X, together with the data described above, is a countably-complete join semilattice on 1.

Let Y be another countably-complete join semilattice on 1. Notice that Y is also a ωcppo on 1. Therefore, by the universal property of X, there exists a unique ωcppo morphism between X and Y. It is not difficult to prove that the latter is also a countably-complete join semilattice morphism and, moreover, the only existing one. □

Notice that the free ωcppo on a general X is not the free countably-complete join semilattice on X since it does not have binary joins. We noticed this already in Section 3.4.1 when we introduced the binary opera-

tion race on $\mathsf{D}\,X$. As a consequence, the majorization of a stream presented in the proof of Proposition 3.4 is not definable.

From Theorem 3.3 and Proposition 3.4, we have that S is isomorphic to $\mathsf{D}\,1/{\approx}$, under the assumption of countable choice. Moreover it is isomorphic to $\mathsf{A}\,1$.

We define $\mathsf{P_S}\,X = \sum_{x:\mathsf{S}} (x \equiv \top \to X)$. We show that $\mathsf{P_S}$ carries a monad structure without the requirement of choice principles.

Proposition 3.5. $\mathsf{P_S}$ *is a monad.*

Proof. Notice that $\mathsf{P_S}$ is a functor specified by a container [3]: the set of shapes is $S = \mathsf{S}$, while the set of positions is $P\,x = x \equiv \top$, for all $x : \mathsf{S}$. Therefore, $\mathsf{P_S}$ carries a monad structure if and only if it comes with certain extra structure [8], namely

$$\mathsf{e} : \mathsf{S} \qquad \bullet : \prod_{x:\mathsf{S}} (x \equiv \top \to \mathsf{S}) \to \mathsf{S}$$

$$q_0 : \prod_{x:\mathsf{S}} \prod_{v:x\equiv\top\to\mathsf{S}} x \bullet v \equiv \top \to x \equiv \top$$

$$q_1 : \prod_{x:\mathsf{S}} \prod_{v:x\equiv\top\to\mathsf{S}} \prod_{p:x\bullet v\equiv\top} v\,(q_0\,x\,v\,p) \equiv \top$$

satisfying the equations

$$x \bullet (\lambda\_.\,\mathsf{e}) \equiv x \qquad \mathsf{e} \bullet (\lambda\_.\,x) \equiv x$$

$$(x \bullet v) \bullet (\lambda p.\, w\,(q_0\,x\,v\,p)\,(q_1\,x\,v\,p)) \equiv x \bullet (\lambda p.\, v\,p \bullet w\,p)$$

Notice that, in general, more equalities between positions are required to hold. In our case, these equations are all trivial, since the type $x \equiv \top$ is a proposition, for all $x : \mathsf{S}$.

We take $\mathsf{e} = \top$, while the function $\bullet$ is defined by induction on its first argument:

$$\begin{aligned} \top \bullet v &= v\,\mathsf{refl} \\ \bot \bullet v &= \bot \\ \bigvee s \bullet v &= \bigvee(\lambda n.\, s\,n \bullet v'\,n) \end{aligned}$$

where, in the last row, $v' : \prod_{n:\mathbb{N}} (s\,n \equiv \top \to \mathsf{S})$ is obtained from $v : \bigvee s \equiv \top \to \mathsf{S}$ by noticing that $s\,n \equiv \top$ implies $\bigvee s \equiv \top$, for all $n : \mathbb{N}$. It is not difficult to see that the function $\bullet$ "respects equality", i.e., that terms made equal by the 1-constructors of S have the same image under $\bullet$.

The terms q_0 and q_1 are constructed by induction on their first argument $x : \mathsf{S}$. The equation $\mathsf{e} \bullet (\lambda\_.\,x) \equiv x$ holds definitionally. The other two equations are proved by induction on the argument $x : \mathsf{S}$. □

We can prove that, similarly to the quotiented delay monad, the monad P_S delivers free ωcppos. Instead of countable choice, we have to assume that S is the free ωcppo on 1. We know that for this assumption to hold, it suffices that the free ωcppo on 1 exists, by Proposition 3.4.

Proposition 3.6. *If S is the free ωcppo on 1, then $\mathsf{P}_\mathsf{S}\,X$ is the free ωcppo on X.*

Proof. We construct an ωcppo structure on the type $\mathsf{P}_\mathsf{S}\,X$:

- A partial order relation on $\mathsf{P}_\mathsf{S}\,X$ is constructed as follows:

$$(x_1, f_1) \leq' (x_2, f_2) = \sum_{p : x_1 \leq x_2} \prod_{q : x_1 \equiv \top} f_1\, q \equiv f_2\, (\mathsf{le2equiveta}\, p\, q)$$

  where $\mathsf{le2equiveta} : x_1 \leq x_2 \to x_1 \equiv \top \to x_2 \equiv \top$ is an easy consequence of $\top$ being the maximal element of the relation $\leq$.

- The bottom element is $(\bot, f)$, where $f : \bot \equiv \top \to X$ is the empty function, since the type $\bot \equiv \top$ is empty.

- Let t be a stream increasing wrt. $\leq'$. The function t is of the form $\langle s, f\rangle$, for some stream $s : \mathbb{N} \to \mathsf{S}$ increasing wrt. $\leq$ and some function $f : \prod_{n:\mathbb{N}} s\, n \equiv \top \to X$. The least upper bound of t is computed as $(\bigvee s, f')$, where $f' : \bigvee s \equiv \top \to X$ is constructed as follows. First one proves that from a proof of $\bigvee s \equiv \top$ one has a proof of $\|\sum_{n:\mathbb{N}} s\, n \equiv \top\|$. A function from the latter type to X is given applying the elimination principle of propositional truncation to the term f. This operation can be performed because the function f is constant, i.e., $f\, n\, p \equiv f\, m\, q$ for $n, m : \mathbb{N}$, $p : s\, n \equiv \top$ and $q : s\, m \equiv \top$, and the latter fact is true because the stream s is increasing.

Moreover, there exists a function $h : X \to \mathsf{P}_\mathsf{S}\,X$, given by $h\, x = (\top, \lambda\_.\, x)$. It is not difficult to check that the type $\mathsf{P}_\mathsf{S}\,X$, together with the previous data, is a ωcppo on X.

Next, we are given an arbitrary ωcppo on X, let us call it Y. We have to construct a ωcppo morphism between $\mathsf{P}_\mathsf{S}\,X$ and Y. We give a sketch of this construction. The desired map is defined in two steps. First, we give a proof $p : \prod_{x:\mathsf{S}} (x \equiv \top \to X) \to Y$. Remember that, by hypothesis, the type S is the free ωcppo on 1 and therefore it has an associated induction principle derivable from the freeness property. The term p is constructed using this induction principle applied to $x : \mathsf{S}$[1]. Second, we show that the uncurried version of p is ω-continuous. Moreover, it is the only such map between $\mathsf{P}_\mathsf{S}\,X$ and Y. □

[1]Notice that, by definition, S has another induction principle given by its dependent eliminator. This induction principle is not strong enough to construct the term p and we need to recourse to the stronger induction principle of the free ωcppo on 1.

In the presence of higher inductive inductive types, $\mathsf{A}\,1$ is the free ωcppo on 1. Therefore, the type $\mathsf{P_S}\,X$ is isomorphic to $\mathsf{A}\,X$. As a consequence, assuming countable choice, $\mathsf{P_S}\,X$ is isomorphic to $\mathsf{D}\,X/{\approx}$. We do not show it here, but one can construct the isomorphism between $\mathsf{P_S}\,X$ and $\mathsf{D}\,X/{\approx}$ also directly, without going through $\mathsf{A}\,X$, but still assuming countable choice.

By Proposition 3.5, we know that $\mathsf{P_S}$ is a monad. One can prove a stronger result: $\mathsf{P_S}$ is a partial map classifier in the sense of [72], classifying specifically partial functions with a semidecidable domain of definedness. This means that maps in the Kleisli category of $\mathsf{P_S}$ are in one-to-one correspondence with maps in a category of partial maps. In fact, notice that the following isomorphism holds:

$$(X \to \mathsf{P_S}\,Y) = \left(X \to \sum_{x:\mathsf{S}} (x \equiv \top \to Y)\right) \cong \sum_{f:X\to\mathsf{S}} \left(\left(\sum_{x:X} f\,x \equiv \top\right) \to Y\right)$$

An inhabitant of the last type can be considered as a map between a subtype U of X and Y. The subtype U is of the form $\sum_{x:X} f\,x \equiv \top$ for a certain function $f : X \to \mathsf{S}$. The type S behaves like a type of truth values, where $\top$ corresponds to truth and $\bot$ to falsehood. The function f can then be seen as a predicate over X with values in S. In this sense U corresponds to a subtype of X characterized by the predicate f.

The type S is typically called *Sierpinski set* [45] or *Rosolini's dominance* [79]. It is a fundamental ingredient in the development of synthetic domain theory [57] and synthetic topology [15].

3.6 Summary

In this chapter, we studied Capretta's delay datatype D [26] and its variant $\mathsf{D}_{\approx}$ obtained by quotienting D by weak bisimilarity. In particular, we showed that the monad structure on D does not naively lift to $\mathsf{D}_{\approx}$. We provided three solutions involving the assumption of classical or semi-classical principles, namely the limited principle of omniscience, the axiom of weak countable choice or the axiom of propositional choice. The solution using weak countable choice, presented in Section 3.2.2, is a refinement of a solution obtained using countable choice described in the author's publication [30]. The solution using propositional choice has been suggested to us by Martin Escardó. Our investigation exposes a general issue involving the relation between quotient types and infinite datatypes, such as non-wellfounded trees or well-founded but non-finitely branching trees.

Afterwards, to argue that the need for choice principles to define the multiplication of the quotiented delay monad was not incidental, but pointed to an intrinsic issue, in Section 3.4 we showed an additional construction related to the quotiented delay datatype that requires the assumption of

countable choice. We showed that the type $\mathsf{D}_{\approx}X$ is the free ω-complete pointed partial order over X, under the assumption of countable choice.

Finally, in Section 3.5 we presented a refinement of Altenkirch et al.'s construction of the partiality monad in homotopy type theory [9]. We defined a monad for partiality using standard higher inductive types and we showed that our datatype is isomorphic to Altenkirch et al.'s datatype and to the quotiented delay datatype, assuming countable choice. Our construction is directly implementable in the Coq proof assistant, which currently lack support for inductive-inductive definitions.

The material presented in Sections 3.4 and 3.5 is included in a journal extension of the author's publication [30] currently under revision.

Chapter 4

ω-complete pointed classifying monads

The quotiented delay monad, introduced in Section 3.2, is useful for "modeling partial functions" and "introducing non-termination as an effect" in type theory. In this chapter, we explain in what sense exactly the quotiented delay monad meets these aims. To do so, we introduce the notion of ω-complete pointed classifying monad. A monad like this is first of all a "monad of partiality", that Cockett and Lack have termed a classifying monad [36], in that its Kleisli category is a restriction category whose pure maps are total.

Restriction categories are an abstract axiomatic framework by Cockett and Lack [35] for reasoning about (generalizations of the idea of) partiality of functions. In a restriction category, every map defines an endomap on its domain, the corresponding partial identity map. Restriction categories cover a number of examples of different flavors and are sound and complete with respect to the more concrete partial map categories. A partial map category is based on a given category (of total maps) and a map in it is a map from a subobject of the domain.

A ω-complete pointed classifying monad is a "monad of non-termination" in that its Kleisli category is ω**CPPO**-enriched wrt. the "less defined than" order on homsets induced by the restriction operation. In other words, it is an ω-complete pointed restriction category (in a sense that is analogous to finite-join restriction categories [53]).

We show that the quotiented delay datatype possesses an ω-complete pointed classifying monad structure. This is a consequence of the fact that the quotiented delay datatype delivers free ω-complete pointed partial orders (Section 3.4). From this observation, we further prove that the quotiented delay datatype is the initial ω-complete pointed classifying monad. Intuitively, this tells us that the Kleisli category of this monad is the minimal setting in Martin-Löf type theory for non-terminating functions.

The initiality result is only interesting, if the category of ω-complete pointed classifying monads contains at least some other interesting examples. A class of examples is given by partial map classifiers, specified using

Rosolini's notion of dominance [79]. Some further examples are ω-complete pointed almost-classifying monads where the word 'almost' refers to dropping one of the conditions of a classifying monad. We present two of such monads: the countable powerset monad and the state monad transformer.

This chapter is organized as follows. In Section 4.1, we review partial map categories, restriction categories and the completeness theorem. In Section 4.2, we define ω-complete pointed classifying monads and prove some properties about them. In Section 4.4, we prove that the quotiented delay monad is the initial ω-complete pointed classifying monad. In Section 4.5, we present some other examples of ω-complete pointed classifying monads.

Our discussion on ω-complete pointed classifying monads applies to general categories. The discussion of the delay monad is carried out for **Set**; generalizing it is future work.

In Appendix A, we present the Agda formalization of Section 4.1. The Agda formalization of Sections 4.2, 4.3 and 4.4 is available at `http://cs.ioc.ee/~niccolo/thesis/`.

4.1 The mathematics of partiality

In this section, we present an overview of partial map categories and restriction categories.

4.1.1 Partial map categories

Partial map categories are a concrete approach to partiality. A partial map category is based on some given category whose maps one wants to regard as total.

The idea then is that a partial map is just a total map from a subobject of the domain, the "domain of definedness". It is ok to accept only certain subobjects as domains of definedness. But the collection of acceptable subobjects must satisfy some closure conditions.

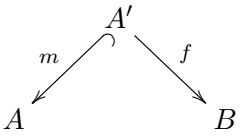
Definition 4.1. A *stable system of monics* for a category $\mathbb{X}$ is a collection $\mathcal{M}$ of monics of $\mathbb{X}$ containing all isomorphisms and closed under composition and arbitrary pullbacks.

(Note that built into this definition is existence of arbitrary pullbacks of monics from $\mathcal{M}$.)

Definition 4.2. Given a category $\mathbb{X}$ and a stable system of monics $\mathcal{M}$ for it, the corresponding *partial map category* $\mathbf{Par}(\mathbb{X}, \mathcal{M})$ is given as follows:

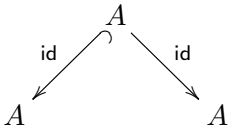
Objects: objects in $\mathbb{X}$.

Maps: a map from A to B is a span



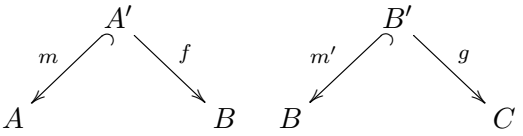
in $\mathbb{X}$, with $m \in \mathcal{M}$.

Identities: identity on A is the span

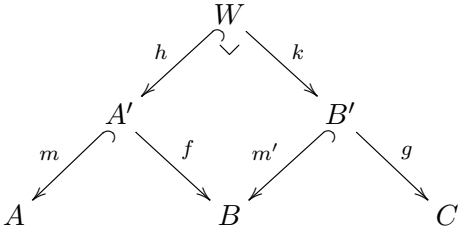


(note that $\mathcal{M}$ contains all isomorphisms, so all identities).

Composition: composition of spans

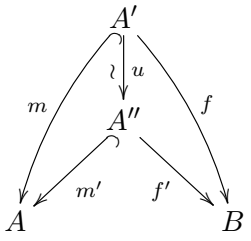


is given in terms of the pullback of f along m' by



(note that $\mathcal{M}$ is closed under arbitrary pullbacks and composition).

Equality of maps in $\mathbf{Par}(\mathbb{X}, \mathcal{M})$ is defined up to isomorphism of spans: two maps (m, f) and (m', f') between two objects A and B are considered equal, if there exists an isomorphism u such that the triangles in the following diagram commute.



(As a consequence, it is unproblematic that pullbacks are uniquely determined only up to isomorphism.)

A map (m, f) is called *total*, if m is an isomorphism.

$\mathbb{X}$ is a subcategory of $\mathbf{Par}(\mathbb{X}, \mathcal{M})$.

4.1.2 Restriction categories

Restriction categories are an axiomatic formulation of categories of "partial functions". Very little is stipulated: any partial function must define a partial endofunction, intuitively the corresponding partial identity function on the domain, satisfying some equational conditions.

Definition 4.3. A *restriction category* is a category $\mathbb{X}$ together with an operation called *restriction* that associates to every $f : A \to B$ a map $\overline{f} : A \to A$ such that

R1 $f \circ \overline{f} \equiv f$

R2 $\overline{g} \circ \overline{f} \equiv \overline{f} \circ \overline{g}$ for all $g : A \to C$

R3 $\overline{g} \circ \overline{f} \equiv \overline{g \circ \overline{f}}$ for all $g : A \to C$

R4 $\overline{g} \circ f \equiv f \circ \overline{g \circ f}$ for all $g : B \to C$

The restriction of a map $f : A \to B$ should be thought of as a "partial identity function" on A, a kind of a specification, in the form of a map, of the "domain of definedness" of f.

A map $f : A \to B$ of $\mathbb{X}$ is called *total*, if $\overline{f} \equiv \mathsf{id}_A$. Total maps define a subcategory $\mathbf{Tot}(\mathbb{X})$ of $\mathbb{X}$.

Definition 4.4. A *restriction functor* between restriction categories $\mathbb{X}$ and $\mathbb{Y}$, with restrictions $\overline{(-)}$ resp. $\widetilde{(-)}$, is a functor F between the underlying categories such that $F\overline{f} \equiv \widetilde{Ff}$.

Restriction categories and restriction functors form a category.

Lemma 4.1. *In a restriction category*

(i) monic maps are total, i.e., $\overline{f} \equiv id_A$ for any monic map $f : A \to B$;

(ii) for any map $f : A \to B$, its restriction $\overline{f}$ is an idempotent, i.e., $\overline{f} \circ \overline{f} \equiv \overline{f}$;

(iii) the restriction operation itself is idempotent, i.e., $\overline{\overline{f}} \equiv \overline{f}$ for any map $f : A \to B$;

(iv) $\overline{g \circ f} \equiv \overline{\overline{g} \circ f}$ for any maps $f : A \to B$ and $g : B \to C$.

Every restriction category is equipped with a partial order: $f \leq g$ if and only if $f \equiv g \circ \overline{f}$. That is, f is less defined than g, if f coincides with g on f's domain of definedness. Notice that, for all $f : X \to Y$, we have $\overline{f} \leq \mathsf{id}_X$.

Lemma 4.2. *In a restriction category* $\mathbb{X}$

(i) the ordering $\leq$ *makes* $\mathbb{X}$ **Poset***-enriched, i.e., for all* $h : W \to X$, $f, g : X \to Y$ *and* $k : Y \to Z$, *if* $f \leq g$, *then* $k \circ f \circ h \leq k \circ g \circ h$;

(ii) if $f \leq g$, *then* $\overline{f} \leq \overline{g}$, *for all* $f, g : X \to Y$.

Example 4.1. The category **Set** of sets and functions (and more generally any category $\mathbb{X}$) is a restriction category with the trivial restriction $\overline{f} = \mathsf{id}$. The category **Par**(**Set**,"all bijections") is isomorphic, as a restriction category, to **Set**.

Example 4.2. The category **Pfn** of sets and partial functions is a restriction category with the restriction

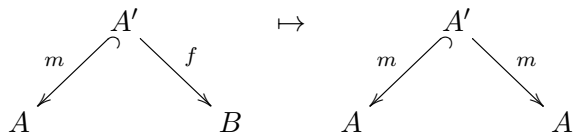
$$\overline{f}(x) = \begin{cases} x & \text{if } f(x) \text{ is defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Pfn is the Kleisli category of the maybe monad defined by $\mathsf{Maybe}\ A = A + 1$. The category **Par**(**Set**,"decidable injections") is isomorphic, as a restriction category, to **Pfn**.

Example 4.3. The subcategory **Prfn** of **Pfn** given by the object $\mathbb{N}$ and all unary partial recursive functions is a restriction category with restriction as defined in Example 4.2. Note that for a partially recursive function its restriction is also partially recursive. No partial map category is isomorphic, as a restriction category, to the restriction category **Prfn**. The reason is that **Prfn** does not have objects for all domains of definition of the maps of **Prfn**, i.e., recursively enumerable sets.

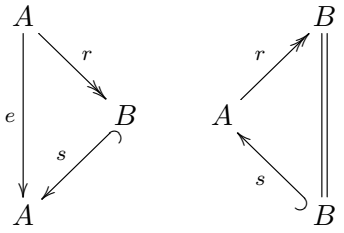
Example 4.4. Other examples are given by Kleisli categories of almost-classifying monads, that we introduce and study later in this chapter. Our central example is the Kleisli category of the quotiented delay monad.

Theorem 4.1. *Any partial map category is a restriction category, with the restriction operation given by*



4.1.3 Idempotents, splitting idempotents

Recall that an endomap $e : A \to A$ is called an *idempotent*, if $e \circ e \equiv e$. It is called a *split idempotent*, if there exists an object B and two arrows $s : B \to A$ (*section*) and $r : A \to B$ (*retraction*) such that the following diagrams commute.

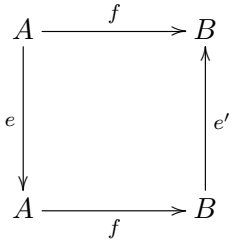


(it is automatic that s is monic and r epic).

Every split idempotent is an idempotent. In the converse direction, idempotents do not always split. But one can take any collection $\mathcal{E}$ of idempotents of a category $\mathbb{X}$ that includes all identity maps and formally split them by moving to another category $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ defined by:

Objects: idempotents from $\mathcal{E}$.

Maps: a map from $e : A \to A$ and $e' : B \to B$ is a map $f : A \to B$ of $\mathbb{X}$ such that



Identities: identity on $e : A \to A$ is e.

Composition: inherited from $\mathbb{X}$.

When $\mathcal{E}$ is the collection of all idempotents of $\mathbb{X}$, the category $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ is known as the Karoubi envelope of $\mathbb{X}$.

$\mathbb{X}$ embeds fully in $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ because the collection $\mathcal{E}$ contains all the identities. Moreover, all idempotents from $\mathcal{E}$ split in $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$: given an idempotent $e : A \to A$ from $\mathcal{E}$, the corresponding map $e : \mathsf{id}_A \to \mathsf{id}_A$ in $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ splits via the object $e : e \to e$ with section $e : e \to \mathsf{id}_A$ and retraction $e : \mathsf{id}_A \to e$.

Lemma 4.3. *Given a restriction category $\mathbb{X}$ and any collection $\mathcal{E}$ of idempotents of $\mathbb{X}$, $\mathbb{X}$ embeds fully, as a restriction category, into $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$, with the restriction of $f : e \to e'$ given by $\widehat{f} = \overline{f} \circ e$.*

In a restriction category $\mathbb{X}$, we call a map $e : A \to A$ a *restriction idempotent*, if $e \equiv \overline{e}$. Lemma 4.1(*ii*) tells us that every restriction idempotent is an idempotent. It follows from Lemma 4.1(*iii*) that restriction idempotents are precisely those maps e for which $e \equiv \overline{f}$ for some f.

$\mathbb{X}$ is called a *split restriction category*, if all of its restriction idempotents split.

Lemma 4.4. *Given a restriction category $\mathbb{X}$, for $\mathcal{R}$ the collection of all restriction idempotents, the restriction category $\mathbf{Split}_{\mathcal{R}}(\mathbb{X})$ is a split restriction category.*

The lemma is proved by observing that every restriction idempotent $f : e \to e$ of $\mathbf{Split}_{\mathcal{R}}(\mathbb{X})$ is a restriction idempotent of $\mathbb{X}$ (as $f \equiv \widehat{f} = \overline{f} \circ e \equiv \overline{f} \circ \overline{e} \equiv \overline{\overline{f} \circ e}$) and therefore an object of $\mathbf{Split}_{\mathcal{R}}(\mathbb{X})$. We can therefore split it via f (as an object) with the section $f : f \to e$ and retraction $f : e \to f$.

Example 4.5. In **Prfn**, restriction idempotents do not split. By splitting the restriction idempotents of **Prfn**, we embed it fully, as a restriction category, into a restriction category $\mathbf{Prfn}^*$, where an object is a recursively enumerable subset of $\mathbb{N}$ and a map between two such sets A and B is a partial recursive function between A and B, by which we mean a partial recursive function $f : \mathbb{N} \to \mathbb{N}$ such that $\mathrm{dom}(f) \subseteq A$ and $\mathrm{rng}(f) \subseteq B$. Its restriction is the corresponding partial identity function on A.

In the subcategory $\mathbf{Tot}(\mathbf{Prfn}^*)$, a map between A and B is a total recursive function between A and B, i.e., a partial recursive function $f : \mathbb{N} \to \mathbb{N}$ such that $\mathrm{dom}(f) = A$ and $\mathrm{rng}(f) \subseteq B$.

4.1.4 Completeness

If all restriction idempotents of a restriction category split, which intuitively means that all domains of definedness are present in it as objects, then the restriction category is a partial map category on its subcategory of total maps.

Theorem 4.2. *Every split restriction category $\mathbb{X}$ is isomorphic, as a restriction category, to a partial map category on the subcategory $\mathbf{Tot}(\mathbb{X})$: for $\mathcal{M}$ the stable system of monics given by the sections of the restriction idempotents of $\mathbb{X}$, it holds that*

$$\mathbb{X} \cong \mathbf{Par}(\mathbf{Tot}(\mathbb{X}), \mathcal{M})$$

From Lemmata 4.3 and 4.4 and Theorem 4.2 we obtain the following corollary.

Corollary 4.1. *A restriction category $\mathbb{X}$ embeds fully, as a restriction category, into a partial map category.*

Example 4.6. The restriction category $\mathbf{Prfn}^*$ is isomorphic, as a restriction category, to $\mathbf{Par}(\mathbf{Tot}(\mathbf{Prfn}^*),$"all total recursive injections"$)$.

4.2 ω-complete pointed classifying monads

In this section, we introduce our monads of non-termination, that we call ω-complete pointed classifying monads. Their definition is built on Cockett and Lack's restriction categories and classifying monads [36] and Cockett and Guo's finite-join restriction categories [53].

4.2.1 Classifying monads

First, some notation. Given a monad $(T, \eta, \_^*)$, we write $\mathbf{Kl}(T)$ for its Kleisli category. We write $g \diamond f$ for the composition $g^* \circ f$ of g and f in $\mathbf{Kl}(T)$.

Definition 4.5. We call a monad T an *almost-classifying monad*, if there exists an operation

$$\frac{f : X \to TY}{\overline{f} : X \to TX}$$

called *restriction*, subject to the following conditions:

CM1 $f \diamond \overline{f} \equiv f$, for all $f : X \to TY$

CM2 $\overline{g} \diamond \overline{f} \equiv \overline{f} \diamond \overline{g}$, for all $f : X \to TY$ and $g : X \to TZ$

CM3 $\overline{g} \diamond \overline{f} \equiv \overline{g \diamond \overline{f}}$, for all $f : X \to TY$ and $g : X \to TZ$

CM4 $\overline{g} \diamond f \equiv f \diamond \overline{g \diamond f}$, for all $f : X \to TY$ and $g : Y \to TZ$

CM5 $\overline{\eta_Y \circ f} \equiv \eta_X$, for all $f : X \to Y$

We call it a *classifying monad*, if it also satisfies

CM6 $\overline{\mathsf{id}_{TX}} \equiv T\eta_X$

In other words, T is an almost-classifying monad, if its Kleisli category $\mathbf{Kl}(T)$ is a restriction category (conditions CM1–CM4) in which pure maps are total (condition CM5). Notice that the condition CM1 is a consequence of CM4 and CM5:

$$f \diamond \overline{f} \equiv f \diamond \overline{\eta_Y \diamond f} \overset{\text{CM4}}{\equiv} \overline{\eta_Y} \diamond f \overset{\text{CM5}}{\equiv} \eta_Y \diamond f \equiv f$$

The additional condition CM6 of a classifying monad was postulated by Cockett and Lack in order to connect classifying monads and partial map classifiers, or more generally, classified restriction categories and classified $\mathcal{M}$-categories (Theorem 3.6 of [36]), $\mathcal{M}$-categories being Robinson

and Rosolini's [77] framework for partiality. While it fulfills this purpose, this condition is very restrictive for other purposes. First of all, it forbids a general monad T from being a classifying monad whose Kleisli category has all maps total. Indeed, define $\overline{f} = \eta_X$, for all $f : X \to TY$. Then conditions CM1–CM5 trivially hold, while condition CM6 is usually false, since generally $\overline{\mathsf{id}_{TX}} = \eta_{TX} \not\equiv T\eta_X$. Moreover, it excludes some other useful monads, as we will see in Section 4.5.2.

Definition 4.6. A *classifying monad morphism* between classifying monads T and S, with restrictions $\overline{(-)}$ resp. $\widetilde{(-)}$, is a monad morphism σ between the underlying monads such that $\sigma \circ \overline{f} \equiv \widetilde{\sigma \circ f}$, for all $f : X \to TY$.

(Almost) classifying monads and (almost) classifying monad morphisms form categories.

An important class of classifying monads is given by the equational lifting monads of Bucalo et al. [25]. Recall that a strong monad T, with left strength ψ, is called *commutative*, if the following diagram commutes [62]:

$$\begin{array}{ccc} T\,X \times T\,Y & \xrightarrow{\psi_{TX,Y}} & T\,(T\,X \times Y) \\ \downarrow{\scriptstyle \phi_{X,TY}} & & \downarrow{\scriptstyle \phi^*_{X,Y}} \\ T\,(X \times T\,Y) & \xrightarrow{\psi^*_{X,Y}} & T\,(X \times Y) \end{array}$$

Here $\phi = T\mathsf{swap} \circ \psi \circ \mathsf{swap}$ is the right strength.

Definition 4.7. An *equational lifting monad* is a commutative monad making the following diagram commute:

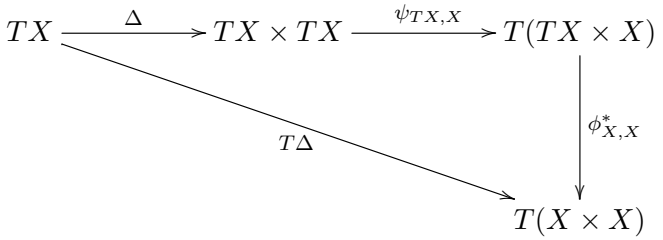
$$\begin{array}{ccc} TX & \xrightarrow{\Delta} & TX \times TX \\ & {\scriptstyle T\langle \eta_X, \mathsf{id}_X \rangle} \searrow & \downarrow{\scriptstyle \psi_{TX,X}} \\ & & T(TX \times X) \end{array} \tag{4.1}$$

Every equational lifting monad is canonically a classifying monad. Its restriction operation is defined with the aid of the strength:

$$\overline{f} = \; X \xrightarrow{\langle \mathsf{id}_X, f \rangle} X \times TY \xrightarrow{\psi_{X,Y}} T(X \times Y) \xrightarrow{T\pi_0} TX$$

Notice that, in order to construct an almost-classifying monad, we can relax condition 4.1 above and consider Cockett and Lack's copy monads [37].

Definition 4.8. A *copy monad* is a commutative monad making the following diagram commute:



The notion of copy monad is equivalent to Jacobs' notion of relevant monad [58]. Every equational lifting monad is a copy monad:

$$\phi^* \circ \psi \circ \Delta \equiv \phi^* \circ T\langle \eta, \mathsf{id} \rangle \equiv (\phi \circ (\eta \times \mathsf{id}) \circ \Delta)^* \equiv (\eta \circ \Delta)^* = T\Delta$$

Every copy monad is canonically an almost-classifying monad. Its restriction operation is defined as the one of equational lifting monads.

4.2.2 ω-joins

Notice that, being a restriction category, the Kleisli category of a classifying monad is **Poset**-enriched in the partial order: $f \leq g$ if and only if $f \equiv g \diamond \overline{f}$ (Lemma 4.2).

Definition 4.9. A classifying monad T is a *ω-complete pointed classifying monad*, if there exist two operations

$$\frac{}{\bot_{X,Y} : X \to TY} \qquad \frac{s : \mathbb{N} \to (X \to TY) \quad \mathsf{isIncr}_{\leq}\, s}{\cup s : X \to TY}$$

satisfying the following conditions:

BOT1 $\bot_{X,Y} \leq f$, for all $f : X \to TY$

BOT2 $\bot_{Y,Z} \diamond f \equiv \bot_{X,Z}$, for all $f : X \to TY$

LUB1 $s\, n \leq \cup s$, for all $n : \mathbb{N}$ and increasing $s : \mathbb{N} \to (X \to TY)$

LUB2 if $s\, n \leq t$ for all $n : \mathbb{N}$, then $\cup s \leq t$, for all $t : X \to TY$ and increasing $s : \mathbb{N} \to (X \to TY)$

LUB3 $\cup s \diamond f \equiv \cup(\lambda n.\, s\, n \diamond f)$, for all $f : X \to TY$ and increasing $s : \mathbb{N} \to (Y \to TZ)$

Conditions BOT1, LUB1 and LUB2 state that every homset in $\mathbf{Kl}(T)$ is a ω-cppo. Conditions BOT2 and LUB3 state that precomposition in $\mathbf{Kl}(T)$ is strict and continuous.

It is actually possible to prove that $\mathbf{Kl}(T)$ is $\omega\mathbf{CPPO}$-enriched. Moreover, the $\bot$ and $\cup$ operations interact well with restriction, as stated in the following lemma.

Lemma 4.5. *Let T be an ω-complete pointed classifying monad. Then the following equalities hold:*

BOT3 $f \diamond \bot_{X,Y} \equiv \bot_{X,Z}$, *for all* $f : Y \to TZ$

BOTR $\overline{\bot_{X,Y}} \equiv \bot_{X,X}$

LUB4 $f \diamond \cup s \equiv \cup(\lambda n.\, f \diamond s\, n)$, *for all* $f : Y \to TZ$ *and increasing* $s : \mathbb{N} \to (X \to TY)$

LUBR $\overline{\cup s} \equiv \cup(\lambda n.\, \overline{s\, n})$, *for all increasing* $s : \mathbb{N} \to (X \to TY)$

Notice that the right-hand sides of LUB3, LUB4 and LUBR are well defined, i.e., the streams that the $\cup$ operation is applied to are chains, thanks to Lemma 4.2.

Definition 4.10. A *ω-complete pointed classifying monad morphism* between ω-complete pointed classifying monads T and S is a classifying monad morphism σ between the underlying classifying monads such that $\sigma \circ \bot \leq' \bot'$ and $\sigma \circ \cup s \leq' \cup' (\lambda n.\, \sigma \circ sn)$, for all increasing $s : \mathbb{N} \to (X \to TY)$.

In the definition above, the least upper bound $\cup(\lambda n.\, \sigma \circ sn)$ is well-defined, since postcomposition with a classifying monad morphism is a monotone operation. In other words, for all $f, g : X \to TY$ with $f \leq g$, we have $\sigma \circ f \leq' \sigma \circ g$. ω-complete pointed classifying monads and ω-complete pointed classifying monad morphisms form a category.

4.2.3 Uniform iteration

If a category is $\omega\mathbf{CPPO}$-enriched, it has an iteration operation that is uniform wrt. all maps. Given a monad T whose Kleisli category is $\omega\mathbf{CPPO}$-enriched, this means that we have an operation

$$\frac{f : X \to T(Y + X)}{f^\dagger : X \to TY}$$

satisfying the conditions

ITE1 $f^\dagger \equiv [\eta_Y, f^\dagger] \diamond f$, for all $f : X \to T(Y + X)$

ITE2 $g \diamond f^\dagger \equiv ([T\mathsf{inl} \circ g, T\mathsf{inr} \circ \eta_X] \diamond f)^\dagger$, for all $f : X \to T(Y + X)$ and $g : Y \to TZ$

ITE3 $(T[\mathsf{id}_{Y+X}, \mathsf{inr}] \circ f)^\dagger \equiv (f^\dagger)^\dagger$, for all $f : X \to T((Y + X) + X)$

ITEU if $f \diamond h \equiv [T\mathsf{inl} \circ \eta, T\mathsf{inr} \circ h] \diamond g$ then $f^\dagger \diamond h \equiv g^\dagger$, for all $f : X \to T(Y + X)$, $g : Z \to T(Y + Z)$ and $h : Z \to T\,X$

Concretely, the operation $(-)^\dagger$ is defined as follows. Let $f : X \to T(Y + X)$. We construct a stream $s : \mathbb{N} \to (X \to TY)$ by

$$s\,0 = \bot_{X,Y} \qquad s\,(n+1) = [\eta_Y, s\,n] \diamond f$$

The stream s is a chain, since the function $\lambda g.\,[\eta_Y, g] \diamond f$ is order-preserving. We define $f^\dagger = \cup s$. That $(-)^\dagger$ satisfies ITE1 is checked as follows. Clearly, $f^\dagger \leq [\eta_Y, f^\dagger] \diamond f$, since $s\,n \leq [\eta_Y, f^\dagger] \diamond f$, for all $n : \mathbb{N}$. For the converse inequality $[\eta_Y, f^\dagger] \diamond f \leq f^\dagger$, it is enough to notice that $[\eta_Y, \cup s] \diamond f \equiv \cup(\lambda n.\,[\eta_Y, s\,n] \diamond f)$ and that $[\eta_Y, s\,n] \diamond f \leq f^\dagger$, for all $n : \mathbb{N}$.

A monad whose Kleisli category has an iteration operation uniform wrt. pure maps is called a complete Elgot monad [50, 51].

4.3 Classifying monad structure on $\mathsf{D}_\approx$

In this section, we see how to construct almost-classifying and classifying monad structures on D and $\mathsf{D}_\approx$. Notice that the monad structure on D described in Section 3.1 does not possess any non-trivial classifying monad structure. Intuitively, the reason lies in the fact that computations which converge to the same value at different paces are not equal. We propose two solutions to this problem.

(i) We change the notion of equality to weak bisimilarity and work with the quotiented delay monad $\mathsf{D}_\approx$ described in Section 3.2. This monad is an equational lifting monad.

(ii) Alternatively, we can change the notion of bind on D. In this way we are able to construct a copy monad structure already on D, without quotienting. Hence, with this modification, D becomes an almost-classifying monad. Still, for obtaining a classifying monad, this does not suffices.

First, we give a proof of (i).

Theorem 4.3. *The monad $\mathsf{D}_\approx$ is an equational lifting monad and therefore a classifying monad.*

Proof. We need to prove that, for all $q : \mathsf{D}_\approx X$, we have $\mathsf{str}_\approx\,(q, q) \equiv \mathsf{D}_\approx\langle \eta_\approx, \mathsf{id}\rangle\,q$, where $\mathsf{str}_\approx$ is the strength operation of $\mathsf{D}_\approx$. Using the induction principle of quotients, it is sufficient to show that, for all $c : \mathsf{D}\,X$, we have $\mathsf{str}_\approx\,([c], [c]) \equiv \mathsf{D}_\approx\langle \eta_\approx, \mathsf{id}\rangle\,[c]$. We know that $\mathsf{str}_\approx\,([c], [c]) \equiv [\mathsf{str}\,([c], c)]$ and $\mathsf{D}_\approx\langle \eta_\approx, \mathsf{id}\rangle\,[c] \equiv [\mathsf{D}\langle \eta_\approx, \mathsf{id}\rangle\,c]$. We show by corecursion on c that $\mathsf{str}\,([c], c) \sim \mathsf{D}\langle \eta_\approx, \mathsf{id}\rangle\,c$:

- if $c = \mathsf{now}\, x$, then both terms are equal to $\mathsf{now}\,([\mathsf{now}\, x], x)$);
- if $c = \mathsf{later}\, c'$, we have to show, after an application of the second constructor of strong bisimilarity, that $\mathsf{str}\,([\mathsf{later}\, c'], c') \sim \mathsf{D}\langle \eta_{\approx}, \mathsf{id}\rangle\, c'$. This is true, since by corecursion we have $\mathsf{str}\,([c'], c') \sim \mathsf{D}\langle \eta_{\approx}, \mathsf{id}\rangle\, c'$ and we know $[c'] \equiv [\mathsf{later}\, c']$. □

In the rest of this chapter, we build on top of Theorem 4.3 and we show that $\mathsf{D}_{\approx}$ is the initial ω-complete pointed classifying monad.

We now move to the proof of (ii). We show how to endow the type D with a copy monad structure without the need of quotienting by weak bisimilarity. The unit is again now, we change the bind operation. In order to have an easy description of this construction, it is convenient to give an alternative presentation of the delay monad. We already showed in Section 3.2.2 that the type $\mathsf{D}\, X$ is isomorphic to the type of streams over $X + 1$ containing at most one element from X (the isomorphism in Equation 3.1). Alternatively, we can say that $\mathsf{D}\, X$ is isomorphic to the type of increasing streams over $X + 1$ with respect to the ordering $\leq_{\mathsf{S}}$ on $X + 1$ inductively defined by the rules:

$$\frac{}{\mathsf{inl}\, x \leq_{\mathsf{S}} \mathsf{inl}\, x} \qquad \frac{}{\mathsf{inr}\, * \leq_{\mathsf{S}} \mathsf{inl}\, x}$$

So we define the type $\mathsf{DS}\, X = \sum_{s:\mathbb{N}\to X+1} \mathsf{isIncr}_{\leq_{\mathsf{S}}}\, s$, isomorphic to $\mathsf{D}\, X$. Notice that the stream functor $\mathsf{Stream}\, X = \mathbb{N} \to X$ is a monad. The unit returns a constant stream, while the bind operation on a function $f : X \to \mathsf{Stream}\, Y$ and a stream $s : \mathsf{Stream}\, X$ returns the diagonal of the stream of streams $[f\,(s\,0), f\,(s\,1), f\,(s\,2), \dots]$. The existence of a distributive law $l_X : (\mathsf{Stream}\, X) + 1 \to \mathsf{Stream}\,(X + 1)$ between the stream monad and the maybe monad induces a monad structure on the functor $\mathsf{Stream}_{+1}\, X = \mathsf{Stream}\,(X + 1)$. Concretely, its unit and bind operations can be described as follows:

$$\begin{aligned}
&\eta_{\mathsf{S}} : X \to \mathsf{Stream}_{+1}\, X\\
&\eta_{\mathsf{S}}\, x\, n = \mathsf{inl}\, x\\
&\\
&\mathsf{bind}_{\mathsf{S}} : (X \to \mathsf{Stream}_{+1}\, Y) \to \mathsf{Stream}_{+1}\, X \to \mathsf{Stream}_{+1}\, Y\\
&\mathsf{bind}_{\mathsf{S}}\, f\, s\, n = \mathsf{case}\ s\, n\ \mathsf{of}\\
&\qquad \mathsf{inl}\, x \mapsto f\, x\, n\\
&\qquad \mathsf{inr}\, * \mapsto \mathsf{inr}\, *
\end{aligned}$$

It is easy to see that $\eta_{\mathsf{S}}\, x$ is increasing wrt. $\leq_{\mathsf{S}}$, for all $x : X$. Moreover, given a function $f : X \to \mathsf{DS}\, Y$ and an increasing stream $s : \mathsf{Stream}_{+1}\, X$, the stream $\mathsf{bind}_{\mathsf{S}}\,(\mathsf{fst} \circ f)\, s$ is also increasing. Therefore, DS inherits the monad structure from Stream_{+1}.

Since the types $\mathsf{DS}\,X$ and $\mathsf{D}\,X$ are isomorphic, we also described a monad structure on D. Intuitively, the new bind operation on D, that we call $\mathsf{bind}_\wedge$, acts on a function $f : X \to \mathsf{D}\,Y$ and a computation $c : \mathsf{D}\,X$ as follows: if $c = \mathsf{never}$, then $\mathsf{bind}_\wedge\, f\, c = \mathsf{never}$; if $c \downarrow x$, then $\mathsf{bind}_\wedge\, f\, c = c \wedge f\, x$, where the meet operation $\wedge$ is corecursively defined with the help of the auxiliary operation $\wedge'$:

$$
\begin{aligned}
&\wedge' : \mathsf{D}\,X \to \mathsf{D}\,Y \to \mathsf{D}\,(X \times Y)\\
&(\mathsf{now}\,x) \wedge' (\mathsf{now}\,y) = \mathsf{now}\,(x, y)\\
&(\mathsf{now}\,x) \wedge' (\mathsf{later}\,c_2) = \mathsf{later}\,((\mathsf{now}\,x) \wedge' c_2)\\
&(\mathsf{later}\,c_1) \wedge' (\mathsf{now}\,y) = \mathsf{later}\,(c_1 \wedge' (\mathsf{now}\,y))\\
&(\mathsf{later}\,c_1) \wedge' (\mathsf{later}\,c_2) = \mathsf{later}\,(c_1 \wedge' c_2)
\end{aligned}
$$

$$
\begin{aligned}
&\wedge : \mathsf{D}\,X \to \mathsf{D}\,Y \to \mathsf{D}\,Y\\
&c_1 \wedge c_2 = \mathsf{D}\,\mathsf{snd}\,(c_1 \wedge' c_2)
\end{aligned}
$$

The meet of two computations $\mathsf{later}^k\,(\mathsf{now}\,x)$ and $\mathsf{later}^n\,(\mathsf{now}\,y)$ is given by $\mathsf{later}^{\max(k,n)}\,(\mathsf{now}\,y)$. Notice the difference between $\mathsf{bind}_\wedge$ and the operation bind introduced in Section 3.1: if $c = \mathsf{later}^k\,(\mathsf{now}\,x)$ and $f\,x = \mathsf{later}^n\,(\mathsf{now}\,y)$, then $\mathsf{bind}_\wedge\, f\, c = \mathsf{later}^{\max(k,n)}\,(\mathsf{now}\,y)$, while $\mathsf{bind}\, f\, c = \mathsf{later}^{k+n}\,(\mathsf{now}\,y)$.

Theorem 4.4. *The monad* $(\mathsf{D}, \mathsf{now}, \mathsf{bind}_\wedge)$ *is a copy monad and therefore an almost-classifying monad.*

Proof. We need to prove that, for all $c : \mathsf{D}\,X$, we have $\mathsf{costr}^*_\wedge(\mathsf{str}_\wedge\,(c, c)) \equiv \mathsf{D}\Delta\,c$, where $\mathsf{str}_\wedge$ and $\mathsf{costr}_\wedge$ are the left and right strength operations associated to the monad $(\mathsf{D}, \mathsf{now}, \mathsf{bind}_\wedge)$. It is not difficult to show that the functions $\mathsf{costr}_\wedge \diamond \mathsf{str}_\wedge$ and $\mathsf{D}\Delta$ are both propositionally equal to $\wedge'$. □

4.4 $\mathsf{D}_\approx$ is the initial ω-complete pointed classifying monad

We extend the order $\sqsubseteq_\approx$ from Section 3.4.2 to maps in $\mathbf{Kl}(\mathsf{D}_\approx)$ in the usual pointwise way. Namely, let $f, g : X \to \mathsf{D}_\approx\,Y$, we say that $f \sqsubseteq_\approx g$ if and only if, for all $x : X$, $f\,x \sqsubseteq_\approx g\,x$. (Notice that we use the same notation $\sqsubseteq_\approx$ for functions as well).

Lemma 4.6. *For all* $f, g : X \to \mathsf{D}_\approx\,Y$*, we have* $f \sqsubseteq_\approx g$ *if and only if* $f \leq g$ *(where* $\leq$ *is the restriction order).*

Theorem 4.5. *The monad* $\mathsf{D}_\approx$ *is an* ω*-complete pointed classifying monad.*

Proof. Let X and Y be two types. The bottom element of the homset $X \to \mathsf{D}_{\approx} Y$ is the constant map $\lambda\_.\,[\mathsf{never}]$. Let $s : \mathbb{N} \to (X \to \mathsf{D}_{\approx} Y)$ be an increasing stream wrt. $\leq$. We define

$$\begin{aligned} &\cup_{\approx} s : X \to \mathsf{D}_{\approx} Y \\ &(\cup_{\approx} s)\, x = \omega\mathsf{race}_{\approx}\, (\lambda n.\, s\, n\, x)\, p \end{aligned}$$

where p is a proof that the stream $\lambda n.\, s\, n\, x$ is increasing wrt. $\sqsubseteq_{\approx}$, which is the case thanks to Lemma 4.6.

One now should verify that conditions BOT1, BOT2 and and LUB1–LUB3 are satisfied. Conditions BOT1, LUB1 and LUB2 follow directly from $\mathsf{D}_{\approx} Y$ being a ωcppo, as described in Section 3.4. Conditions BOT2 and LUB3 follow from the fact that $\mathsf{bind}_{\approx}$ (the bind operation of $\mathsf{D}_{\approx}$) is a strict and continuous function between $X \to \mathsf{D}_{\approx} Y$ and $\mathsf{D}_{\approx} X \to \mathsf{D}_{\approx} Y$.

□

Let T be a ω-complete pointed almost-classifying monad. We already noted that the type $X \to T\, Y$ is a ωcppo, for all types X and Y. In particular, every type $T\, X \cong 1 \to T\, X$ is a ωcppo. Explicitly, given $x_1, x_2 : T\, X$, we define $x_1 \leq x_2$ as $(\lambda *.\, x_1) \leq (\lambda *.\, x_2)$. The bottom element of $T\, X$ is $\bot_{1,X}\, *$, while the join of a chain $s : \mathbb{N} \to T\, X$ is $\cup(\lambda n \lambda *.\, s\, n)\, *$.

We show that there is a unique ω-complete pointed almost-classifying monad morphism between $\mathsf{D}_{\approx}$ and T. This characterizes the quotiented delay monad as the universal monad of non-termination.

Theorem 4.6. $\mathsf{D}_{\approx}$ *is the initial* ω*-complete pointed almost-classifying monad (and therefore also the initial* ω*-complete pointed classifying monad).*

Proof. Let (T, η, bind) be a ω-complete pointed almost-classifying monad. Since $T\, X$ is a ωcppo, there is a unique ωcppo morphism between $\mathsf{D}_{\approx} X$ and $T\, X$. Therefore, we define

$$\begin{aligned} &\sigma : \mathsf{D}_{\approx} X \to T\, X \\ &\sigma = \widehat{\eta} \end{aligned}$$

Remember from Section 3.4.2 that $\widehat{\eta}$ is the unique ωcppo morphism such that $\widehat{\eta} \circ \eta_{\approx} \equiv \eta$.

First, we show that σ is a monad morphism:

- $\sigma \circ \eta_{\approx} \equiv \eta$ by the universal property of the free ωcppo.
- Given $f : X \to \mathsf{D}_{\approx} Y$, we have $\sigma \circ \mathsf{bind}_{\approx} f \equiv \mathsf{bind}\, (\sigma \circ f) \circ \sigma$, because both maps are ωcppo morphisms between $\mathsf{D}_{\approx} X$ and $T\, Y$ and both maps are equal to $\sigma \circ f$ when precomposed with $\eta_{\approx}$.

Second, we show that σ is an almost-classifying monad morphism. We have to show that $\sigma \circ \overline{f} \equiv \widetilde{\sigma \circ f}$ for all $f : X \to \mathsf{D}_{\approx} Y$. Notice that, for all $x : 1 \to X$, we have:

$$\sigma \circ \overline{f} \circ x \overset{\text{CM4}}{\equiv} \sigma \circ \mathsf{D}_{\approx} x \circ \overline{f \circ x} \overset{\text{nat}}{\equiv} T\, x \circ \sigma \circ \overline{f \circ x}$$
$$\widetilde{\sigma \circ f} \circ x \overset{\text{CM4}}{\equiv} T\, x \circ \widetilde{\sigma \circ f \circ x}$$

Therefore it is sufficient to show $\sigma \circ \overline{c} \equiv \widetilde{\sigma \circ c}$ for all $c : 1 \to \mathsf{D}_{\approx} X$. The maps $g\, c = \sigma \circ \overline{c}$ and $h\, c = \widetilde{\sigma \circ c}$ are both strict and continuous maps of type $(1 \to \mathsf{D}_{\approx} X) \to (1 \to T\, 1)$, and the latter type is isomorphic $\mathsf{D}_{\approx} X \to T\, 1$. Notice that, since $\mathsf{D}_{\approx} X$ is the free ωcppo over X, we know that there exists only one strict and continuous map between $\mathsf{D}_{\approx} X$ and $T\, 1$ that sends terminating computations to $\eta *$. Notice that, for all $x : 1 \to X$, we have

$$g\,(\eta_{\approx X} \circ x) = \sigma \circ \overline{\eta_{\approx X} \circ x} \overset{\text{CM5}}{\equiv} \sigma \circ \eta_{\approx 1} \equiv \eta_1$$
$$h\,(\eta_{\approx X} \circ x) = \widetilde{\sigma \circ \eta_{\approx X} \circ x} \equiv \widetilde{\eta_X \circ x} \overset{\text{CM5}}{\equiv} \eta_1$$

This shows that $g \equiv h$, and therefore σ is a classifying monad morphism.

Finally, σ is a ω-complete pointed almost-classifying monad morphism since $\widehat{\eta}$ is a ωcppo morphism between $\mathsf{D}_{\approx} X$ and $T\, X$. In particular, it preserves the bottom elements and it is continuous.

It remains to check that σ is the unique ω-complete pointed almost-classifying monad morphism between $\mathsf{D}_{\approx}$ and T. Let τ be another ω-complete pointed almost classifying monad morphism between $\mathsf{D}_{\approx}$ and T. In particular, for all types X, we have that τ is a ωcppo morphism between $\mathsf{D}_{\approx} X$ and $T\, X$ and also $\tau \circ \eta_{\approx} \equiv \eta$. Therefore, by the universal property of the free ωcppo $\mathsf{D}_{\approx}$, we have that $\tau \equiv \widehat{\eta} = \sigma$. □

One might wonder whether $\mathbf{Kl}(\mathsf{D}_{\approx})$ could be the free ω-complete pointed restriction category over **Set**. This is not the case, since the latter has as objects sets and as maps between X and Y elements of $\mathsf{D}_{\approx}(X \to Y)$. This observation is an adaptation of a construction for finite join restriction categories by Grandis described by Guo [53].

Notice that the fundamental ingredient in the proof of Theorems 4.5 and 4.6 is the fact that $\mathsf{D}_{\approx}$ delivers ωcppos. In fact, one can prove a more general statement without assuming countable choice. Suppose that P is a monad delivering ωcppos. Then P is an equational lifting monad. Moreover, it is the initial ω-complete pointed classifying monad.

One might also wonder if Theorems 4.5 and 4.6 could be further generalized to settings different from **Set**, for example, to Cartesian closed categories with a natural number object, in which it is possible to internalize the notion of ωcppo [12]. We believe that such generalization is possible. It is left for future work.

4.5 Other monads of non-termination

In the previous section, we showed that $\mathsf{D}_\approx$ is the initial ω-complete pointed almost-classifying monad and also the initial ω-complete pointed classifying monad. This would not be a significant result, if the categories of ω-complete pointed classifying and almost-classifying monads were lacking other interesting examples. It is immediate that these categories are non-trivial, since at least the monad $\mathsf{Termin}\, X = 1$ is another ω-complete pointed classifying monad. Since Termin is the final object in the category of monads, it is also the final ω-complete pointed almost-classifying monad and the final ω-complete pointed classifying monad. But of course we are looking for more interesting examples.

4.5.1 Dominances and partial map classifiers

The notion of dominance was introduced by Rosolini in his PhD thesis [79]. In topos theory, dominances are used to characterize the domain of definedness of partial maps, generalizing the notion of subobject classifier. In type theory, the notion of dominance was reformulated by Escardó and Knapp [46]. Here we give a definition which is equivalent to theirs.

Definition 4.11. A type D is a *dominance* if it comes with the following data:

- an injective map $[\![-]\!] : D \to \Omega$;
- an element $1_D : D$;
- an operation $\Sigma_D : \prod_{X:D} ([\![X]\!] \to D) \to D$:
- a proof of $[\![1_D]\!] \equiv 1$;
- a proof of $[\![\Sigma_D\, X\, Y]\!] \equiv \sum_{x:[\![X]\!]} [\![Y\, x]\!]$, for all $X : D$ and $Y : [\![X]\!] \to D$.

The type Ω is the type of all propositions, $\Omega = \sum_{X:\mathcal{U}} \mathsf{isProp}\, X$. Given $X : \Omega$, we simply write $x : X$ meaning $x : \mathsf{fst}\, X$. Remember that we are assuming proposition extensionality, whose formal definition is given in Section 2.1.1. This means that $X \equiv 1$ is equivalent to say that the proposition X is inhabited.

Here are some examples of dominances.

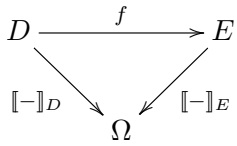
Example 4.7. The initial dominance is the unit type 1, while the final dominance is Ω. Other important examples of dominances include the type of booleans Bool and the Sierpinski type S introduced in Section 3.5. Assuming propositional choice, the type $\mathsf{D}_\approx 1$ is also a dominance (this is essentially the proof of Theorem 3.4). In all these examples, the interpretation morphism is defined as $[\![X]\!] = (X \equiv 1_D)$.

Example 4.8. Every Lawvere-Tierney topology [65, Ch. 5.1] specifies a dominance. Remember that a Lawvere-Tierney topology is given by an endomap $j : \Omega \to \Omega$ such that

- $j\,1 \equiv 1$;
- $j\,(j\,X) \equiv j\,X$;
- $j\,(X \times Y) \equiv j\,X \times j\,Y$.

The dominance associated to j is given by the type of j-closed truth values $\sum_{X:\Omega} j\,X \equiv X$. A standard example of Lawvere-Tierney topology is the double negation operator $\neg\neg$. More generally, every monad on Ω specifies a dominance in a similar way.

Definition 4.12. A *dominance morphism* between two dominances D and E is a function $f : D \to E$ such that the following triangle commutes:



A dominance morphism preserves the dominance structure: $f\,1_D \equiv 1_E$ and $f\,(\Sigma_D\,X\,Y) \equiv \Sigma_E\,(f\,X)\,Y'$, for $X : D$ and $Y : [\![X]\!]_D \to D$, where Y' is defined as follows:

$$\begin{aligned} &Y' : [\![f\,X]\!]_E \to E \\ &Y' = \mathsf{subst}\,(\lambda a.\, a \to E)\,e\,(f \circ Y) \end{aligned}$$

and $e : [\![f\,X]\!]_E \equiv [\![X]\!]_D$ follows from f being a dominance morphism.

Given a dominance D, we define the *partial map classifier* [72]:

$$\mathsf{P}_D X = \sum_{U:D} [\![U]\!] \to X$$

This is a monad, thanks to the dominance laws. Moreover, it is not difficult to prove that it is an equational lifting monad and therefore a classifying monad.

Example 4.9. (i) $\mathsf{P}_1 X \cong X$, i.e. P_D is the identity monad.

(ii) $\mathsf{P}_{\mathsf{Bool}} X \cong X + 1$, i.e. P_D is the maybe monad.

(iii) P_{S} is the partial map classifier introduced in Section 3.5.

(iv) By Proposition 3.3, $\mathsf{P}_{\mathsf{D}_{\approx 1}} X \cong \mathsf{D}_{\approx} X$.

(v) A non-example: the type $\mathsf{Termin}\, X = 1$ is isomorphic to $\sum_{*:1} 0 \to X$. The latter looks similar to a partial map classifier, but it is not. In fact, for $D = 1$ and $[\![*]\!] = 0$, we have that D is not a dominance, since $[\![1_D]\!] = [\![*]\!] \not\equiv 1$.

Clearly, not all monads P_D are ω-complete pointed classifying monads. The main example of this phenomenon is the maybe monad. In order to construct the join of a chain $s : \mathbb{N} \to X + 1$, we need to decide whether there exist an element $x : X$ and a number $k : \mathbb{N}$ such that $s\, k = \mathsf{inl}\, x$, or $s\, n = \mathsf{inr}\, *$ for all $n : \mathbb{N}$. This decision requires LPO. As discussed in Section 3.2.1, the assumption of LPO is sufficient to show that $\mathsf{Maybe}\, X$ and $\mathsf{D}_{\approx} X$ are isomorphic types.

It turns out that the monad P_D is a ω-complete pointed classifying monad if the dominance D is closed under countable joins wrt. the partial order $X \leq Y = [\![X]\!] \to [\![Y]\!]$.

Definition 4.13. A dominance D is a *countably-complete dominance* if it comes with the following data:

- an element 0_D;
- an operation $\bigvee_D : (\mathbb{N} \to D) \to D$;
- a proof $[\![0_D]\!] \equiv 0$;
- a proof $[\![\bigvee_D S]\!] \equiv \| \sum_{n:\mathbb{N}} [\![S\, n]\!] \|$.

A countably-complete dominance is also called "a dominance for which the type $\mathbb{N}$ is overt" [15]. It is not difficult to show that 0_D is the bottom element of D, while $\bigvee_D S$ is the least upper bound of the stream S.

Notice that a dominance morphism $f : D \to E$ also preserves the countably-complete dominance structure: $f\, 0_D \equiv 0_E$ and $f\, (\bigvee_D S) \equiv \bigvee_E (f \circ S)$, for $S : \mathbb{N} \to D$.

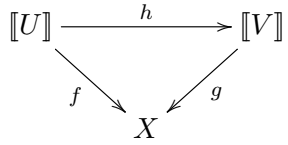
Remember that the type S of Section 3.5 is the free countably-complete join semilattice on 1 by construction. Notice that every countably-complete dominance D is a countably-complete join semilattice on 1. Therefore there exists a unique countably-complete join semilattice morphism f between S and D. It is not difficult to show that f is a dominance morphism (this is proved using the induction principle of S). This shows that S is the initial countably-complete dominance. In other words, we recovered in our framework the fact that the Sierpinski type is the smallest dominance such that the type $\mathbb{N}$ is overt [15].

Given a countably-complete dominance D, the monad P_D is a ω-complete pointed classifying monad. This is because every type $\mathsf{P}_D X$ is a ωcppo. To

see this, we first define a partial order on $\mathsf{P}_D X$:

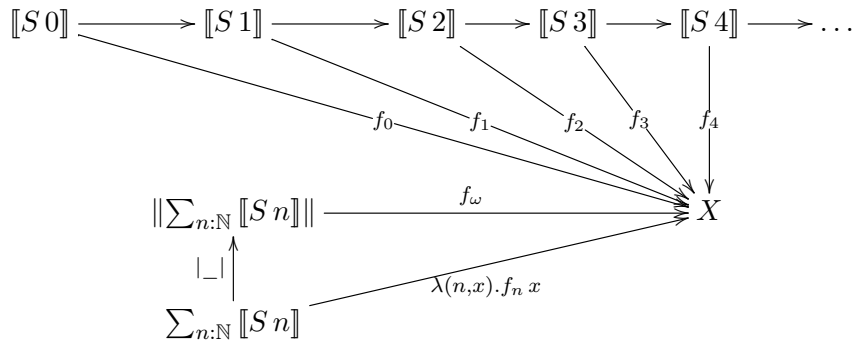
$$(U, f) \leq (V, g) = \sum_{h:[\![U]\!] \to [\![V]\!]} \prod_{x:X} f\, x \equiv g\,(h\, x)$$

That is, two elements in $\mathsf{P}_D X$ are related by $\leq$, if there exists a morphism in the slice category of X connecting their images under $[\![-]\!]$:



In other words, the poset $(\mathsf{P}_D X, \leq)$ is the full subcategory of the slice category of X in which objects are propositions in D. If D is a countably-complete dominance, this poset is also a ωcppo. The bottom element is given by 0_D (the only inhabitant of the type $[\![0_D]\!] \to X$ is the empty function, since $[\![0_D]\!]$ is the empty type).

Let $F : \mathbb{N} \to \mathsf{P}_D X$ be an increasing stream. Therefore $F = \langle S, f \rangle$, for some $S : \mathbb{N} \to D$ and $f : \prod_{n:\mathbb{N}} [\![S\, n]\!] \to X$. We write f_n instead of $f\, n$. We define the least upper bound $\cup F = (\bigvee_D S, f_\omega)$, where f_ω is computed as the following colimit:



The function f_ω is obtained as the lifting of $\lambda(n, x).f_n\, x : \sum_{n:\mathbb{N}} [\![S\, n]\!] \to X$. Notice that the latter function is constant, i.e., $f_n\, x_n \equiv f_m\, x_m$ for all $x_n : [\![S\, n]\!]$ and $x_m : [\![S\, m]\!]$. In fact, suppose w.l.o.g. $m \leq n$. Then, since the stream is increasing, there exists a function $h : [\![S\, m]\!] \to [\![S\, n]\!]$ such that $f_m\, x_m \equiv f_n\,(h\, x_m)$. The type $[\![S\, n]\!]$ is a proposition, therefore $h\, x_m \equiv x_n$, which in turns implies $f_m\, x_m \equiv f_n\, x_n$.

As usual, the order $\leq$ extends to function spaces. The monad P_D is an equational lifting monad. It is not difficult to show that this is equivalent to the restriction order and that P_D satisfies the laws of an ω-complete pointed classifying monad. Theorem 4.6 tells us that, under the assumption of countable choice, there exists a unique ω-complete pointed classifying monad morphism between $\mathsf{D}_\approx$ and P_D for any countably-complete dominance D.

4.5.2 Countable powerset monad

An example of an ω-complete pointed almost-classifying monad that is not a classifying monad (since the condition CM6 is not met), is given by the countable powerset construction. This monad is typically employed to model non-deterministic computations. We introduced the countable powerset of a type X as the higher inductive type $\mathcal{P}_\infty X$ in Section 3.5. Another possible implementation is the following:

$$\mathcal{P}'_\infty X = \mathsf{Stream}\,(X + 1)/\mathsf{SameElem}$$

where $\mathsf{SameElem}\,s_1\,s_2 = \prod_{x:X} x \in s_1 \leftrightarrow x \in s_2$ and $x \in s = \sum_{n:\mathbb{N}} s\,n \equiv \mathsf{inl}\,x$. The types $\mathcal{P}_\infty X$ and $\mathcal{P}'_\infty X$ are isomorphic under the assumption of countable choice.

Intuitively, the restriction $\overline{f}$ of a map $f : X \to \mathcal{P}_\infty Y$ is the map that, given $x : X$, returns the singleton $\{x\}$, if $f\,x$ is non-empty, and returns the empty set otherwise. Notice that the restriction order on $\mathcal{P}_\infty X \cong 1 \to \mathcal{P}_\infty X$ is therefore different from set inclusion. In fact, for $s_1, s_2 : \mathcal{P}_\infty X$, intuitively $s_1 \leq s_2$ if and only if $s_1 \equiv s_2$ or $s_1 \equiv \emptyset$.

4.5.3 State monad transformer

New ω-complete pointed almost-classifying monads can be constructed from already constructed ones with the state monad transformer. Recall that the state monad is defined as $\mathsf{State}\,X = S \to X \times S$, where S is a fixed set of states. Given an ω-complete pointed almost-classifying monad T, the functor $\mathsf{StateT}\,T$ defined by $\mathsf{StateT}\,T\,X = S \to T(X \times S)$ is another ω-complete pointed almost-classifying monad. All operations of $\mathsf{StateT}\,T$ are defined in terms of the operations of T. For example, the restriction operation is constructed in the following way:

$$\widetilde{(\_)} : (X \to S \to T(Y \times S)) \to X \to S \to T(X \times S)$$
$$\widetilde{f} = \mathsf{curry}\,(\overline{\mathsf{uncurry}\,f})$$

The bottom and least upper bound operations are constructed analogously.

4.6 Summary

In this chapter, we gave a precise mathematical characterization of the quotiented delay monad as a monad for non-termination. First, in Section 4.2, we introduced ω-complete pointed classifying monads, a refinement of Cockett and Lack's classifying monads [36] in which we ask the restriction order to possess a bottom element and a join operation for increasing streams. In Section 4.4 we proved the main result of this thesis: the quotiented delay

monad $\mathsf{D}_\approx$ is the initial ω-complete pointed classifying monads. This shows that $\mathsf{D}_\approx$ is canonical universal among monads for non-termination in type theory.

The latter result is made meaningful by the existence of other interesting examples of ω-complete pointed classifying monads in type theory, that we presented in Section 4.5, notably partial map classifiers [72] specified by countably-complete dominances.

Another result described in this chapter is the construction of an almost-classifying monad structure on the (unquotiented) delay datatype D given in Section 4.3. This shows that Capretta's delay datatype, when endowed with an optimized monad structure, is already a monad for partiality without the need of quotienting by weak bisimilarity.

The material presented in this chapter is currently unpublished. It has been submitted to a conference.

Chapter 5

Conclusions

In this thesis we continued the study of non-termination in type theory following Capretta's approach [26]. It is a well known fact that the delay monad D is the free completely iterative monad over the identity functor [4], and therefore it is canonical among monads capturing guarded iteration. The delay monad also captures unguarded iteration up to weak bisimilarity. If we wish to capture unguarded iteration up to propositional equality, a natural solution is obtained by quotienting the delay datatype by weak bisimilarity, as suggested by Uustalu in a talk given in 2005 [27].

When trying to formally prove the latter statement, we have to be precise and specify what we mean by "quotienting", since type theory does not possess quotient types. Extending the theory with inductive-like quotients à la Hofmann [54], we found out that $\mathsf{D}_\approx$ cannot be shown to be a monad without assuming additional classical or semi-classical principles, such as LPO, weak countable choice or propositional choice. More generally, this exposes a bad interaction between quotient types and infinite datatypes (non-wellfounded or well-founded but infinitely branching trees).

Capretta [26] showed that using the delay datatype one can encode general recursion in type theory. One would expect this to lift unproblematically to the quotiented delay monad, but this again requires the assumption of LPO or countable choice. Similarly, we need such extra principles when proving that the quotiented delay monad is the lifting monad delivering free ωcppos. This shows that the quotiented delay monad is not a very useful tool for the development of computability theory and domain theory in type theory, if it is not used in combination with additional non-fully constructive principles, such as countable choice.

Nonetheless, we showed that the monad $\mathsf{D}_\approx$ is a partial map classifier, classifying partial maps with semidecidable domain of definedness. We proved the latter result using restriction categories, Cockett and Lack's categorical framework for partiality [35]. This spawned a formalization of the theory of restriction categories in Agda [1, 74]. The main effort in our formalization is the completeness theorem, stating intuitively that every

restriction category with enough domain of definedness is a partial map category.

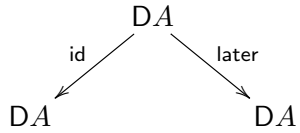
Our successive goal was to show that $\mathsf{D}_\approx$ captures unguarded iteration. Monads that do that are called complete Elgot monads. We were indeed able to prove that $\mathsf{D}_\approx$ possesses a complete Elgot monad structure. Goncharov et al. [51] proved that the maybe monad is the initial complete Elgot monad, if the base category is hyperextensive [5]. We first tried to replicate the latter initiality proof in a constructive setting, switching from Maybe to $\mathsf{D}_\approx$, but without success.

We then aimed at a more specific characterization of the quotiented delay monad. We introduced the notion of ω-complete pointed classifying monad, i.e., classifying monads whose Kleisli categories are ω**CPPO**-enriched wrt. the restriction order. We showed that the monad $\mathsf{D}_\approx$ is the initial ω-complete pointed classifying monad, a consequence of the fact that $\mathsf{D}_\approx$ delivers free ωcppos. The initiality result is meaningful because the class of ω-complete pointed classifying monads is non-trivial, since it also contains other partial map classifiers specified by ω-dominances.

5.1 Future work

5.1.1 Quotiented delay datatype in general categories

The main topic of this thesis is the study of the delay monad and the quotiented delay monad in Martin-Löf type theory. The construction of the delay monad can be performed in more generality in a category with coproducts $\mathbb{C}$ for which the functor $F_A X = A + X$ admits a final coalgebra for all objects A, i.e., $\mathsf{D}A = \nu X.A + X$. The construction of the monad structure on D is straightforward. The quotiented delay monad can hypothetically be constructed in several ways. One idea could be to follow the type-theoretic development of Chapter 3. We can introduce the following relation:

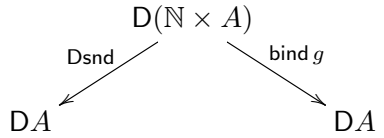


If the base category $\mathbb{C}$ has all finite colimits, we can then take the coequalizer of the above parallel maps:

$$\mathsf{D}A \rightrightarrows \mathsf{D}A \longrightarrow \mathsf{D}_\approx A$$

Proving that $\mathsf{D}_\approx$ is a monad is not straightforward, as one ends up in troubles similar to the ones discussed in Section 3.2. This means that the construction of a monad structure on $\mathsf{D}_\approx$ also requires the assumption of classical or semi-classical principles.

Alternatively, we could directly take the above coequalizer in the category of monads over $\mathbb{C}$. But this cannot be done, since later is not a monad morphism. Therefore, we consider a different relation:



where $g : \mathbb{N} \times A \to \mathsf{D}A$ is the unique (co)algebra morphism between the initial and the final coalgebra of F_A (notice that $\mathbb{N} \times A \cong \mu X.A + X$). Notice that $G\,A = \mathbb{N} \times A$ is a monad, and $\mathsf{D} \circ G$ is a monad since there exists a distributive law $\mathbb{N} \times \mathsf{D}A \to \mathsf{D}(\mathbb{N} \times A)$ given by the strength operation of D. In this way, $\mathsf{D}_{\approx}$ would be a monad by construction. A general treatment of transfinite constructions, including coequalizers of monads, has been developed by Kelly [61].

Another further possible direction of research is the generalization of Theorems 4.5 and 4.6 to settings different from **Set**. For example, in cartesian closed categories with a natural number object, in which it is possible to internalize the notion of ωcppo [12].

5.1.2 Partiality in homotopy type theory

We set the development of the theory of the quotiented delay monad in Martin-Löf type theory, but we could have alternatively developed our formalization in homotopy type theory [83], more specifically in 0-truncated homotopy type theory, as already stated in Section 2.1.1. In fact, function extensionality and proposition extensionality are consequences of the univalence axiom, while working with 0-truncated types, i.e. sets, simulates uniqueness of identity proofs. We still need to postulate that bisimilar coinductive data is equal.

There exist several libraries for homotopy type theory in dependently typed programming languages such as Coq (UniMath, HoTT library), Agda (HoTT-Agda library) and Lean. Future work will include formalizing the Sierpinski space, defined as the higher inductive type S of Section 3.5, in one of these libraries, and formally proving that the partial map classifier P_{S} is isomorphic to Altenkirch et al.'s partiality monad [9]. Our implementation has the advantage of employing only standard higher inductive types and does not require inductive-inductive definitions, which at the moment are not available in the Coq proof assistant.

5.1.3 Formalizing restriction categories, continued

As presented in Appendix A, we have formalized in Agda the first chapters of the theory of restriction categories and the connection between the latter

and partial map categories, whose theory is described in Section 4.1. The next natural step would be to formalize classifying monads and partial map classifiers, and prove similar soundness and completeness results [36]. We also have an Agda formalization of classifying monads, following the theory presented in Chapter 4. But this formalization only includes classifying monads on **Set**. Therefore future work will involve extending the latter to general categories.

Another possible extension of the restriction categories library would be in the direction of restriction categories with additional structure (products, meets, iteration operators, etc.) [37] and Turing categories [33].

5.1.4 Initial complete Elgot monad

As already discussed in the beginning of this chapter, we failed to replicate in type theory Goncharov et al.'s construction of the initial complete Elgot monad in hyperextensive categories [51]. A complete Elgot monad is a monad T that comes with an unguarded iteration operation

$$\frac{f : X \to T(Y + X)}{f^\dagger : X \to TY}$$

which is uniform wrt. pure maps. It is indeed possible to construct a morphism of type $\xi : \mathsf{D}X \to TX$. First we define the the following composite map:

$$\mathsf{D}X \xrightarrow{[\mathsf{now},\mathsf{later}]^{-1}} X + \mathsf{D}X \xrightarrow{\quad\eta\quad} T(X + \mathsf{D}X)$$

where $[\mathsf{now}, \mathsf{later}]^{-1}$ is the final coalgebra structure of $\mathsf{D}X$, which is the inverse of $[\mathsf{now}, \mathsf{later}]$, and η is the unit of T. We define $\xi = (\eta \circ [\mathsf{now}, \mathsf{later}]^{-1})^\dagger$. Problems arise when trying to lift the function ξ to the quotient $\mathsf{D}_{\approx}X$. In fact, we have to provide a compatibility proof of $c_1 \approx c_2 \to \xi\, c_1 \equiv \xi\, c_2$. We do not know how to construct such a proof.

Future work will involve understanding the constructive content of Goncharov et al.'s initiality proof. This would hopefully guide us in proving that the quotiented delay monad is the initial complete Elgot monad on **Set** constructively (under reasonable semi-classical principles). If the latter proof fails, we should come up with a different construction. Our idea would be to define a monad delivering complete Elgot algebras [7]. Recently, Goncharov et al. [52] also declared finding the initial complete Elgot monad in a constructive setting an open problem.

References

[1] The Agda Team: The Agda wiki (2015) `http://wiki.portal.chalmers.se/agda/`

[2] Abel, A., Chapman, J.: Normalization by evaluation in the delay monad: a case study for coinduction via copatterns and sized types. In: Levy, P., Krishnaswami, N. (eds.) Proc. of 5th Wksh. on Mathematically Structured Functional Programming, MSFP 2014, Electron. Proc. in Theor. Comput. Sci., v. 153, pp. 51–67, Open Publishing Assoc. (2014)

[3] Abbott, M., Altenkirch, T., Ghani, N.: Containers: Constructing strictly positive types. Theor. Comput. Sci., 342(1), 3–27 (2005)

[4] Aczel, P., Adámek, J., Milius, S. and Velebil, J.: Infinite trees and completely iterative theories: a coalgebraic view. Theor. Comput. Sci., 300(1–3), 1–45 (2003)

[5] Adámek, J., Borger, R., Milius, S., Velebil, J.: Iterative algebras: how iterative are they? Theory Appl. Cat., 19, 61–92 (2008)

[6] Adámek, J., Milius, S., Velebil, J.: Free iterative theories: a coalgebraic view. Math. Struct. Comput. Sci., 13(2), 259–320 (2003)

[7] Adámek, J., Milius, S., Velebil, J.: Elgot algebras. Log. Meth. Comput. Sci., 2(5), article 4 (2006)

[8] Ahman, D., Chapman, J., Uustalu, T.: When is a container a comonad? Log. Meth. Comput. Sci., v. 10, n. 3, article 14 (2014)

[9] Altenkirch, T., Danielsson, N. A., Kraus, N.: Partiality, revisited: The partiality monad as a quotient inductive-inductive type. In: Esparza, J., Murawski, A. eds., Proc. of 20th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2017, Lect. Notes in Comput. Sci., v 10203, pp. 534–549. Springer (2017)

[10] Awodey, S.: Category Theory. Oxford Logic Guides, Oxford University Press (2006)

[11] Axelsen, H. B., Kaarsgaard, R.: Join inverse categories as models of reversible recursion. J. Log. Algebr. Meth. Program., 87, 33–50 (2017)

[12] Barr, M.: Fixed points in Cartesian closed categories. Theor. Comput. Sci., 70, 65–72 (1990)

[13] Barthe, G., Capretta, V., Pons, O.: Setoids in type theory. J. Funct. Program., 13(2), 261–293 (2003)

[14] Bauer, A.: First steps in synthetic computability theory. Electron. Notes Theor. Comput. Sci., 155, 5–31 (2006)

[15] Bauer, A., Lesnik, D.: Metric spaces in synthetic topology. Ann. Pure Appl. Logic, 163(2), 87–100 (2012)

[16] Bauer, A., Gross, J., Lumsdaine, P.L., Shulman, M., Sozeau, M., Spitters, B.: The HoTT library: a formalization of homotopy type theory in Coq. In: Proc. of 6th ACM SIGPLAN Conf. on Certified Programs and Proofs, CPP 2017, pp. 164-172, ACM (2017)

[17] Benton, N., Kennedy, A., Varming, C.: Some domain theory and denotational semantics in Coq. In Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) Proc. of 22nd Int. Conf. on Theorem Proving in Higher Order Logics, TPHOLs 2009, Lect. Notes in Comput. Sci., v. 5674, pp. 115–130, Springer (2009)

[18] Bishop, E.: Foundations of constructive analysis. McGraw-Hill (1967)

[19] Bove, A.: Another look at function domains. Electron. Notes Theor. Comput. Sci., 249, 61–74 (2009)

[20] Bove, A., Capretta, V.: Recursive functions with higher order domains. In: Urzyczyn, P. (ed.) Proc. of 7th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2005, Lect. Notes in Comput. Sci., v. 3461, pp. 116–130, Springer (2005)

[21] Bove, A., Capretta, V.: Modelling general recursion in type theory. Math. Struct. in Comput. Sci., 15(4), 671–708 (2005)

[22] Bove, A., Capretta, V.: Computation by prophecy. In: Ronchi Della Rocca, S. (ed.), Proc. of 8th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2007, Lect. Notes in Comput. Sci., v. 4583, pp. 70–83, Springer (2007)

[23] Bove, A., Krauss, A., Sozeau, M.: Partiality and recursion in interactive theorem provers - an overview. Math. Struct. in Comput. Sci., 26(1), 38–88 (2016)

[24] Bridges, D. S., Richman, F., Schuster, P.: A weak countable choice principle. Proc. Amer. Math. Soc., 128(9), 2749–2752 (2000)

[25] Bucalo, A., Führmann, C., Simpson, A.: An equational notion of lifting monad. Theor. Comput. Sci., 294(1–2), 31–60 (2003)

[26] Capretta, V.: General recursion via coinductive types. Log. Methods Comput. Sci., 1(2), article 1 (2005)

[27] Capretta, V., Altenkirch, T., Uustalu, T.: Partiality is an effect. Slides for a talk given by Uustalu at the 22nd meeting of IFIP Working Group 2.8 (2005)

[28] Carboni, A.: Bicategories of partial maps. Cah. Top. Geom. Diff., 28, 111–126 (1987)

[29] Cazanescu, V.-E., Stefanescu, G.: Feedback, iteration and repetition. In: Paun, G. (ed.) Mathematical Aspects of Natural and Formal Languages, pp. 43–62, World Scientific (1995)

[30] Chapman, J., Uustalu T., Veltri, N.: Quotienting the delay monad by weak bisimilarity. In: Leucker, M., Rueda, C., Valencia, F. D. (eds.) Proc. of 12th Int. Coll. on Theoretical Aspects of Computing, ICTAC 2015, Lect. Notes in Comput. Sci., v. 9399, pp. 110–125, Springer (2015)

[31] Chapman, J., Uustalu, T., Veltri. Formalizing Restriction Categories. J. Formalized Reasoning, 10(1), 1–36 (2017)

[32] Chicli, L., Pottier, L., Simpson, C.: Mathematical quotients and quotient types in Coq. In: Geuvers, H., Wiedijk, F. (eds.) Selected Papers from Int. Wksh. on Types for Proofs and Programs, TYPES 2002, Lect. Notes in Comput. Sci., v. 2646, pp. 95–107, Springer (2003)

[33] Cockett, J. R. B., Hofstra, P. J. W.: Introduction to Turing categories. Ann. Pure Appl. Logic, 156(2–3), 183–209 (2008)

[34] Cockett, J. R. B., Díaz-Boïls, J., Gallagher, J., Hrubes, P.: Timed sets, complexity, and computability. Electron. Notes in Theor. Comput. Sci., 286, 117–137, Elsevier (2012)

[35] Cockett, J. R. B., Lack, S.: Restriction categories I: categories of partial maps. Theor. Comput. Sci. 270(1–2), 223–259 (2002)

[36] Cockett, J. R. B., Lack, S.: Restriction categories II: partial map classification. Theor. Comput. Sci., 294(1–2), 61–102 (2003)

[37] Cockett, J. R. B., Lack, S.: Restriction categories III: colimits, partial limits, and extensivity. Math. Struct. in Comput. Sci., 17(4), 775–817 (2007)

[38] Cohen,C.: Pragmatic quotient types in Coq. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) Proc. of 4th Int. Conf. on Interactive Theorem Proving, ITP 2013, Lect. Notes in Comput. Sci., v. 7998, pp. 213–228, Springer (2013)

[39] Coquand, T., Mannaa, B., Ruch, F.: Stack semantics of type theory. In: Proc. of 32th Ann. ACM/IEEE Symp. on Logic in Computer Science, LICS 2017, to appear (2017)

[40] Curien, P.-L., Obtulowicz, A: Partiality, cartesian closedness, and toposes. Inform. Comput., 80, 50–95 (1989)

[41] Danielsson, N. A.: Operational semantics using the partiality monad. In: Thiemann, P., Findler, R. B. (eds.) Proc. of 17th ACM SIGPLAN Inte. Conf. on Functional Programming, ICFP 2012, pp. 127–138, ACM (2012)

[42] Di Paola, R. A., Heller, A.: Dominical categories: recursion theory without elements. J. Symb. Log., 52(3), 594–635 (1987)

[43] Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. J. Symb. Log., 65(2), 525–549 (2000)

[44] Elgot, C. C., Bloom, S. L., Tindell, R.: On the algebraic structure of rooted trees. J. Comput. Syst. Sci., 16(3), 361–399 (1978)

[45] Escardó, M.: Synthetic topology of data types and classical spaces. Electron. Notes Theor. Comput. Sci., 87, 21–156, (2004)

[46] Escardó, M. H., Knapp, C. M.: Partial elements and recursion via dominances in univalent type theory. Preprint (2017) `http://www.cs.bham.ac.uk/~mhe/papers/partial-elements-and-recursion.pdf`

[47] Fiore, M. P.: Axiomatic domain theory in categories of partial maps. Distinguished Dissertation Series, Cambridge University Press (1996)

[48] Fiore, M. P., Plotkin, G. D. , Power, A. J.: Complete cuboidal sets in axiomatic domain theory. In: Proc. of 12th Ann. IEEE Symp. on Logic in Computer Science, LICS 1997, pp. 268-278, IEEE Computer Society (1997)

[49] Fourman, M. P., Ščedrov, A.: The "world's simplest axiom of choice" fails. Manuscripta Math. 38(3), 325–332 (1982)

[50] Goncharov, S., Milius, S., Rauch, C.: Complete Elgot monads and coalgebraic resumptions. Electron. Notes Theor. Comput. Sci., 325, 147–168 (2016)

[51] Goncharov, S., Rauch, C., Schröder, L.: Unguarded recursion on coinductive resumptions. Electron. Notes Theor. Comput. Sci., 319, 183–198 (2015)

[52] Goncharov, S., Schröder, L., Rauch, C., Piróg, M.: Unifying guarded and Unguarded iteration. In: Esparza, J., Murawski, A. eds., Proc. of 20th Int. Conf. on Foundations of Software Science and Computation Structures, FoSSaCS 2017, Lect. Notes in Comput. Sci., v 10203, pp. 517-533. Springer (2017)

[53] Guo, X.: Products, joins, meets and ranges in restriction categories. PhD thesis, University of Calgary (2012)

[54] Hofmann, M.: Extensional constructs in intensional type theory. CPHS/BCS Distinguished Dissertations, Springer (1997)

[55] Hughes, J.: Generalising monads to arrows. Sci. Comput. Program., 37(1–3), 67–111 (2000)

[56] Hasegawa, M.: Models of sharing graphs: A categorical semantics of let and letrec. PhD thesis ECS-LFCS-97-360, University of Edinburgh (1997)

[57] Hyland, J. M. E.: First steps in synthetic domain theory. In: Carboni, A., Pedicchio, M. C., Rosolini, G. (eds.) Proc. of Int. Conf. on Category Theory, CT '90, Lecture Notes in Mathematics, v. 1488, pp. 131–156, Springer (1991)

[58] Jacobs, B.: Semantics of weakening and contraction. Ann. Pure Appl. Logic, 69(1), 73–106 (1994)

[59] Jacobs, B., Heunen, C., Hasuo, I.: Categorical semantics for arrows. J. Funct. Program., 19(3–4), 403–438 (2009)

[60] Joyal, A., Street R., Verity, D.: Traced monoidal categories. Math. Proc. Cambridge Phil. Soc., 119(3), 447–468 (1996)

[61] Kelly, M.: A unified treatment of transfinite constructions for free algebras, free monoids, colimits, associated sheaves and so on. Bull. Austral. Math. Soc., 22(1), 1–83 (1980)

[62] Kock, A.: Strong functors and monoidal monads. Arch. Math., 23(1), 113–120 (1972)

[63] Kraus, N., Escardó, M., Coquand, T., Altenkirch, T.: Generalizations of Hedberg's Theorem. In: Hasegawa, M. (ed.) Proc. of 11th Int. Conf. on Typed Lambda Calculi and Applications, TLCA 2013, Lect. Notes in Comput. Sci., v. 7941, pp. 173–188, Springer (2013)

[64] Mac Lane, S.: Categories for the working mathematician, Graduate Texts in Mathematics, v. 5, Springer, 2nd ed. (1998)

[65] Mac Lane, S., Moerdijk, I.: Sheaves in geometry and logic - a first introduction to topos theory. Universitext, Springer (1992)

[66] Maietti, M. E.: About effective quotients in constructive type theory. In: Altenkirch, T., Naraschewski, W., Reus, B. (eds.) Selected Papers from Int. Wksh. on Types for Proofs and Programs, TYPES 1998, Lect. Notes in Comput. Sci., v. 1657, pp. 166–178, Springer (1999)

[67] Martin-Löf, P.: 100 years of Zermelo's axiom of choice: what was the problem with it? Comput. J., 49(3), 345–350 (2006)

[68] Megacz, A.: A coinductive monad for Prop-bounded recursion. In: Stump, A., Xi, H. (eds.) Proc. of ACM Wksh. on Programming Languages meets Program Verification, PLPV 2007, pp. 11–20, ACM (2007)

[69] Menger, K.: An axiomatic theory of functions and fluents. In: Henkin, L., Suppes, P., Tarski, A. (eds.) The Axiomatic Method, Studies Logic Found. Math., v. 27, pp. 454–473, North-Holland (1959)

[70] Milius, S., Litak, T.: Guard your daggers and traces: on the equational properties of guarded (co-)recursion. In: Baelde, D., Carayol, A. (eds.) Proc. of Wksh. on Fixed Points in Computer Science, FICS 2013, Electron. Proc. in Theor. Comput. Sci. 126, pp. 72–86, Open Publishing Assoc. (2016)

[71] Moggi, E.: Notions of computation and monads. Inform. Comput., 93(1), 55–92 (1991)

[72] Mulry, P. S.: Partial map classifiers and partial Cartesian closed categories. Theor. Comput. Sci., 136(1), 109–123 (1994)

[73] Nordström, B., Petersson, K., Smith, J. M.: Programming in Martin-Löf's type theory: an introduction. Oxford University Press (1990)

[74] Norell, U.: Dependently typed programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, S. D. (eds.) Revised Lectures from 6th Int. School on Advanced Functional Programming, AFP 2008, Lect. Notes in Comput. Sci., v. 5832, pp. 230–266, Springer (2009)

[75] Nuo, L.: Quotient types in type theory. PhD thesis, University of Nottingham (2015)

[76] Richman, F.: Constructive mathematics without choice. In: Schuster, P., Berger, U., Osswald, H. (eds.) Proc. of Symp. Reuniting the Antipodes - Constructive and Nonstandard Views of the Continuum, Synthese Library, v. 306, pp. 199–205, Springer (2001)

[77] Robinson, E., Rosolini, G.: Categories of partial maps. Inform. Comput., 79(2), 95–130 (1988)

[78] Rosolini, R.: Domains and dominical categories. Riv. Mat. Univ. Parma, 11, 387–397 (1985)

[79] Rosolini, G.: Continuity and effectiveness in topoi. D.Phil thesis, Oxford University (1986)

[80] Setzer, A.: Partial recursive functions in Martin-Löf type theory. In: Beckmann, A., Berger, U., Löwe, B., Tucker, J. V. (eds.) Proc. of 2nd Conf. on Computability in Europe, CiE 2006, Lect. Notes in Comput. Sci., v. 3988, pp. 505–515 (2006)

[81] Simpson, A., Plotkin, G.: Complete axioms for categorical fixed-point operators. In: Proc. of 15th Ann. IEEE Symp. on Logic in Computer Science, LICS 2000, pp. 30–41, IEEE Computer Society (2000)

[82] Troelstra, A. S., Van Dalen, D.: Constructivism in mathematics: an introduction, v. I. Studies in Logic and the Foundations of Mathematics, v. 121, North-Holland (1988)

[83] The Univalent Foundations Program: Homotopy type theory: univalent foundations of mathematics. Institute for Advanced Study (2013) `http://homotopytypetheory.org/book`

[84] Veltri, N.: Two set-based implementations of quotients in type theory. In: Nummenmaa, J., Sievi-Korte, O., Mäkinen, E. (eds.) Proc. of 14th Symposium on Programming Languages and Software Tools, SPLST 2015, CEUR Workshop Proceedings, v. 1525, pp. 194–205, CEUR-WS.org (2015)

Appendices

Appendix A

Formalizing restriction categories

We now illustrate how we formalized the mathematics presented in Section 4.1 in the Agda proof assistant [1, 74]. We have developed our own library of basic utilities: implementation of quotient types, definitions of categories, functors, monics, isomorphisms, sections, idempotents and pullbacks; proofs of various properties about them, e.g., the pasting lemmas for pullbacks. The main part of the formalization consists of definitions of restriction categories, partial map categories; proofs of important lemmata; proof of the soundness Theorem 4.1; construction of splitting of idempotents; proof of the completeness Theorem 4.2.

We represent algebraic-like structures such as categories as dependent records with fields for the data of the structure and fields for the laws. Typically in our formalization record types are opened before their projection functions are utilized. For example in Section A.2, in the definition of `Functor`, the field `Hom` from the record type `Cat` is in scope and it takes a category as its first argument. Sometimes we open only one specific record. For example, in Section A.3, we fix a particular category `X`. In that section, the field `Hom` is in scope and it corresponds to homsets in `X`. We always specify which terms are in context in a section or in a paragraph. Therefore, the reader should not find difficulties in understanding how to interpret fields of records in different situations.

It is common practice in type theory (and Agda) to use setoids to represent the homset when representing categories, so that the laws are given in term of the equivalence relation of the setoid. For this particular formalization using setoids would be especially heavy as we would require setoids of objects as well as homsets: when constructing the category $\mathbf{Split}_{\mathcal{E}}(\mathbb{X})$ we take objects to be idempotent maps from the class $\mathcal{E}$ in the underlying category $\mathbb{X}$. We instead use inductive-like quotient types, introduced in 2.2.

In Agda, unfinished parts of a definition are denoted by a question mark `?`. In this chapter, we leave some definitions incomplete. We omit some definitions due to reasons of space and/or readability. The full formalization contains no unfinished parts.

The Agda formalization of this chapter is available at `http://cs.ioc.ee/~niccolo/thesis/`. The material presented in this chapter is based on a paper written by the author together with James Chapman and Tarmo Uustalu [31].

A.1 Quotients

We show how we formalize quotient types in Agda. Similar formalizations have been performed in Coq [32, 38].

We define a record type `Quotient` for a type `X` and an equivalence relation `R` on `X` using the standard library machinery for equivalence relations. An equivalence relation on a type `X` is a binary relation on `X` together with a proof of it being reflexive, symmetric and transitive (this predicate is called `isEquivalence` in Agda's standard library).

```
EqR : Set → Set
EqR X = Σ (X → X → Set) IsEquivalence
```

The `Quotient` record type has a field `Q` for the type of equivalence classes of `X` and a field `box` for the canonical projection map `X → Q`. As well as `box` we have a dependent eliminator `lift`, which lifts dependent functions from `X` to functions from `Q`. This operation can only lift `compat`ible functions and hence it takes a compatibility proof as an extra argument. `compat` is a predicate on functions from `X` stating that the function takes related arguments in `X` to equal results. Notice that `box` is compatible by `sound`. Axiom `liftβ` states that applying a lifted function to an abstracted argument is the same as applying the function to the argument directly. The last two fields `liftL` and `liftLβ` correspond to the large dependent eliminator and its computation rule.

```
record Quotient (X : Set)(EQ : EqR X) : Set₁ where
  open Σ EQ renaming (proj₁ to R)
  field
    Q   : Set
    box : X → Q

  compat : (Y : Q → Set)(f : (x : X) → Y (box x)) → Set
  compat Y f = {x₁ x₂ : X} → R x₁ x₂ → f x₁ ≡ f x₂

  field
    sound  : compat _ box
    lift   : (Y : Q → Set)(f : (x : X) → Y (box x))
      (p : compat Y f)(q : Q) → Y q
    liftβ  : (Y : Q → Set)(f : (x : X) → Y (box x))
```

```
    (p : compat Y f)(x : X) → lift Y f p (box x) ≡ f x
  liftL  : (Y : Q → Set₁)(f : (x : X) → Y (box x))
    (p : compat Y f)(q : Q) → Y q
  liftLβ : (Y : Q → Set₁)(f : (x : X) → Y (box x))
    (p : compat Y f)(x : X) → liftL Y f p (box x) ≡ f x
```

Every type together with an equivalence relation on it gives rise to a quotient. This is what we need to postulate in Agda. The record type `Quotient` gives a specification of a quotient, `quot` assumes that this specification holds (is inhabited) for any type and equivalence relation on it.

```
postulate
  quot : (X : Set)(EQ : EqR X) → Quotient X EQ
```

A.2 Categories

Categories are described as a record type with fields for the set of objects, the set of maps between two objects, for any object an identity map and for any pair of suitable maps their composition. Further to this, we have three fields for the laws of a category given as propositional equalities between maps.

```
record Cat : Set₁ where
  field
    Obj  : Set
    Hom  : Obj → Obj → Set
    iden : ∀{A} → Hom A A
    comp : ∀{A B C} → Hom B C → Hom A B → Hom A C
    idl  : ∀{A B}{f : Hom A B} → comp iden f ≡ f
    idr  : ∀{A B}{f : Hom A B} → comp f iden ≡ f
    ass  : ∀{A B C D}{f : Hom C D}{g : Hom B C}{h : Hom A B}
      → comp (comp f g) h ≡ comp f (comp g h)
```

Functors are also described as a record type with fields for the mapping of objects, the mapping of morphisms and the two laws stating that the latter must preserve identities and composition.

```
record Fun (X Y : Cat) : Set where
  field
    OMap  : Obj X → Obj Y
    HMap  : ∀{A B} → Hom X A B → Hom Y (OMap A) (OMap B)
    fid   : ∀{A} → HMap (iden X {A}) ≡ iden Y {OMap A}
    fcomp : ∀{A B C}{f : Hom X B C}{g : Hom X A B}
      → HMap (comp X f g) ≡ comp Y (HMap f) (HMap g)
```

The identity functor has identity maps as mapping of objects and mapping of morphisms, and reflexivity proves the functor laws.

```
idFun : {X : Cat} → Fun X X
idFun = record{
  OMap = id;
  HMap = id;
  fid = refl;
  fcomp = refl}
```

The properties of functors being full and faithful are given as predicates on functors.

```
Full : {X Y : Cat}(F : Fun X Y) → Set
Full {X}{Y} F =
  ∀{A B}{f : Hom Y (OMap F A) (OMap F B)}
    → Σ (Hom X A B) λ g → HMap F g ≡ f

Faithful : {X Y : Cat}(F : Fun X Y) → Set
Faithful {X} F =
  ∀{A B}{f g : Hom X A B} → HMap F f ≡ HMap F g → f ≡ g
```

A.3 Monics, isomorphisms and pullbacks

In this section, we work in a particular category `X`. In Agda, this corresponds to working in a module parameterized by a category `X`. Moreover, as already specified in the beginning of the chapter, we open the specific record `X`. This implies that, for example, the projections `Obj`, `Hom` and `iden` refer to objects, homsets and identity morphisms in the category `X`.

A.3.1 Monic maps

A map `f` is monic if for any suitable maps `g` and `h`, if `comp f g ≡ comp f h` then `g ≡ h`.

```
Mono : ∀{A B}(f : Hom A B) → Set
Mono f = ∀{C}{g h : Hom C _} → comp f g ≡ comp f h → g ≡ h
```

We prove a lemma `idMono` stating that every identity map is monic. An equational proof starts with the word `proof` and ends with the symbol ■. The proof is an alternating sequence of expressions and justifications. It is very close to how one would write it on paper, but we do not gloss over minor details such as appeals to associativity of composition in a category. We must also be very precise about where in an expression we apply a rewrite rule.

```
idMono : ∀{A} → Mono (iden {A})
idMono {g = g}{h} p =
  proof
  g
  ≡⟨ sym idl ⟩
  comp iden g
  ≡⟨ p ⟩
  comp iden h
  ≡⟨ idl ⟩
  h
  ■
```

A.3.2 Isomorphisms

Isomorphism is defined as a predicate on maps which is witnessed by a suitable inverse map and proofs of the two isomorphism properties.

```
record Iso {A B : Obj}(f : Hom A B) : Set where
  field
    inv  : Hom B A
    rinv : comp f inv ≡ iden {B}
    linv : comp inv f ≡ iden {A}
```

We prove that any identity map is trivially an isomorphism, the proof arguments are given by the left identity property `idl` (right identity `idr` also works) of the category `X`.

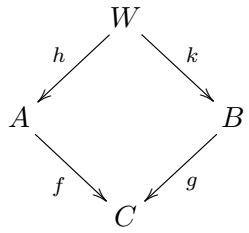
```
idIso : ∀{A} → Iso (iden {A})
idIso = record{
  inv = iden;
  rinv = idl;
  linv = idl}
```

A.3.3 Pullbacks

The definition of pullback is divided into three parts. First we give the definition of a square over a cospan, i.e., a pair of maps `f : Hom A C` and `g : Hom B C` with the same target object. It is a record consisting of an object `W`, two maps `h` and `k` completing the square, and a proof `scom` that the square commutes.

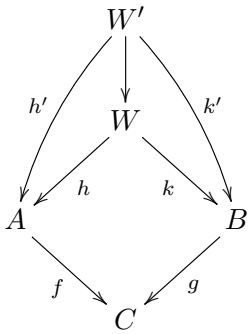
```
record Square {A B C}(f : Hom A C)(g : Hom B C) : Set where
  field
    W    : Obj
    h    : Hom W A
```

```
    k    : Hom W B
    scom : comp f h ≡ comp g k
```



Then we define a map between two squares, called a `SqMap`. It consists of a map `sqMor` between the respective `W` objects of the squares together with proofs of commutation of the two triangles that the map `sqMor` generates.

```
record SqMap {A B C : Obj}{f : Hom A C}{g : Hom B C}
             (sq' sq : Square f g) : Set where
  field
    sqMor   : Hom (W sq') (W sq)
    leftTr  : comp (h sq) sqMor ≡ h sq'
    rightTr : comp (k sq) sqMor ≡ k sq'
```

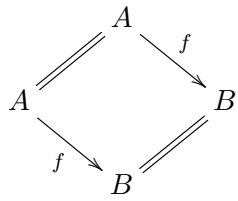


A pullback of maps `f` and `g` consists of a square `sq` and a universal property `uniqPul`: for any other square `sq'` over `f` and `g` there exists a unique map between `sq` and `sq'`.

```
record Pullback {A B C}(f : Hom A C)(g : Hom B C) : Set where
  field
    sq      : Square f g
    uniqPul : (sq' : Square f g)
      → Σ (SqMap sq' sq)
          λ u → (u' : SqMap sq' sq) → sqMor u ≡ sqMor u'
```

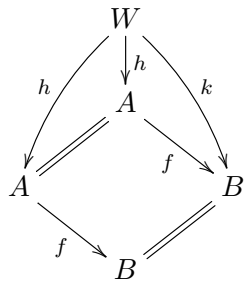
Later we will need two results regarding pullbacks. The first is the definition of a pullback of a map `f` along the identity map. The other two sides completing the pullback square are `f` and the identity map.

```
trivialSquare : ∀{A B}(f : Hom A B) → Square f iden
trivialSquare {A} f = record{
  W = A;
  h = iden;
  k = f;
  scom =
    proof
    comp f iden
    ≅⟨ idr ⟩
    f
    ≅⟨ sym idl ⟩
    comp iden f
    ■}
```



Let `sq` be another square over f and the identity map. We call `W` the object, `h` and `k` the two maps that complete the square `sq`, and `scom` the proof that the square commutes. Then `h` together with the straightforward proofs that the two triangles it generates commute is a map between the two squares.

```
trivialSqMap : ∀{A B}(f : Hom A B)(sq : Square f iden) →
               SqMap sq (trivialSquare f)
trivialSqMap f sq = record{
  sqMor = h sq;
  leftTr = idl;
  rightTr =
    proof
    comp f (h sq)
    ≅⟨ scom sq ⟩
    comp iden (k sq)
    ≅⟨ idl ⟩
    k sq
    ■}
```

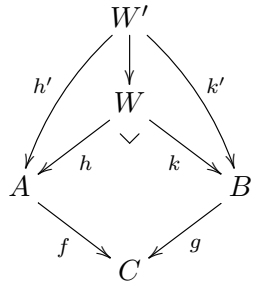


To complete the construction of the pullback, it remains to supply a proof that the map `trivialSqMap f sq` between the arbitrary square `sq` and `trivialSquare f` is unique.

```
trivialPullback : ∀{A B}(f : Hom A B) → Pullback f iden
trivialPullback f = record{
  sq = trivialSquare f;
  uniqPul = λ sq →
    trivialSqMap f sq ,
    λ u →
      proof
      h sq
      ≅⟨ sym (leftTr u) ⟩
      comp iden (sqMor u)
      ≅⟨ idl ⟩
      sqMor u
      ■}
```

The second result regarding pullbacks we need is a theorem stating that any two pullbacks over the same cospan are isomorphic. The isomorphism is the unique map between the two squares provided by the universal property of pullbacks.

```
pullbackIso : ∀{A B C}{f : Hom A C}{g : Hom B C}
              (p p' : Pullback f g) →
              Iso (sqMor (proj₁ (uniqPul p (sq p'))))
pullbackIso p p' = ?
```



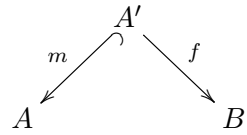
A.4 Partial map categories

Let `X` be a category. A stable system of monics in `X` is a set of maps given by a membership predicate `∈sys` satisfying four properties: every element is monic; all isomorphisms are elements; the set is closed under composition; and the set is closed under pullbacks along arbitrary maps.

```
record StableSys : Set₁ where
  field
    ∈sys     : ∀{A B}(f : Hom A B) → Set
    mono∈sys : ∀{A B}{f : Hom A B} → ∈sys f → Mono f
    iso∈sys  : ∀{A B}{f : Hom A B} → Iso f → ∈sys f
    comp∈sys : ∀{A B C}{f : Hom A B}{g : Hom B C}
      → ∈sys f → ∈sys g → ∈sys (comp g f)
    pul∈sys  : ∀{A B C}(f : Hom A C){m : Hom B C}
      → ∈sys m → Σ (Pullback f m) λ p → ∈sys (h (sq p))
```

A partial map category is a category defined on a stable system of monics `M` on `X`. The objects are the objects of `X` and the maps are spans which are defined as a record type indexed by source and target objects `A` and `B` consisting of a third object `A'`, two maps `mhom` and `fhom` for the left and right leg of the span, and a proof that the left leg `mhom` is a member of the stable system of monics.

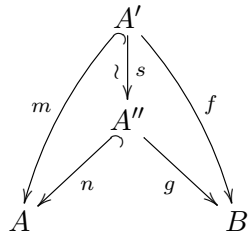
```
record Span (A B : Obj) : Set where
  field
    A'    : Obj
    mhom  : Hom A' A
    fhom  : Hom A' B
    m∈sys : ∈sys mhom
```



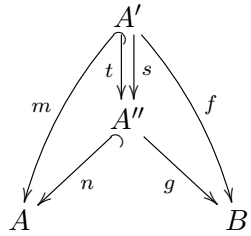
Equality on spans is defined up to isomorphism. We prove the properties of spans up to this isomorphism, and then use a quotient to work with this isomorphism in the place of equality. Two spans `mf` and `ng` are 'equal', if, for some isomorphism between their source objects, the two triangles it generates commute.

```
record _∼Span∼_ {A B}(mf ng : Span A B) : Set where
  field
    s        : Hom (A' mf) (A' ng)
    sIso     : Iso s
```

```
leftTr∼  : comp (mhom ng) s ≡ mhom mf
rightTr∼ : comp (fhom ng) s ≡ fhom mf
```



Notice that, for all `mf ng : Span A B`, the type `mf ∼Span∼ ng` is a proposition, i.e. any two inhabitants of this type are equal. In fact, suppose there are two isomorphisms `s` and `t` between the spans `mf` and `ng`.



In particular, `comp n t ≡ m ≡ comp n s`. Since `n` is monic, we have `s ≡ t`. The relation `_∼Span∼_` forms an equivalence relation.

```
Span∼EqR : ∀{A B} → EqR (Span A B)
Span∼EqR = _∼Span∼_ , ?
```

We quotient `Span A B` by this equivalence relation and we call the result `qspan A B`.

```
qspan : ∀ A B → Quotient (Span A B) Span∼EqR
qspan A B = quot (Span A B) Span∼EqR
```

The carriers `QSpan A B` of such quotients are the maps in the partial map category.

```
QSpan : ∀ A B → Set
QSpan A B = Quotient.Q (qspan A B)
```

We define shorthand names for referring to the quotient machinery for an arbitrary span, e.g.

```
box : {A B : Obj} → Span A B → QSpan A B
box {A}{B} = Quotient.box (qspan A B)
```

Shorthands for the other fields are given in a similar way. From now on the names `compat`, `sound`, `lift` and `liftβ` always refer to the respective fields in `qspan A B`.

The partial map category has sets as objects and `QSpan`s as homsets. Just as homsets are defined in two steps (first `Span`, then `QSpan`), the operations and laws are also defined in two steps. We first define operations on `Span`s and then port them to `QSpan`s. Analogously, we prove the laws up to `_∼Span∼_` and then port them to equality proofs. Note that the whole construction of the partial map category and the soundness proof is performed first up to `_∼Span∼_`. This means that this part of our formalization could be reused even if one wants to take the "setoid approach" to formalizing category theory in type theory.

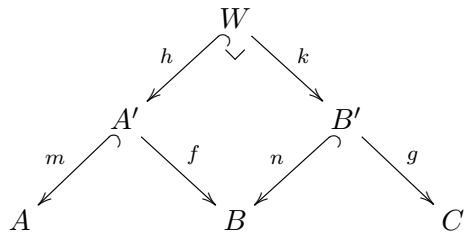
The identity span is trivial to describe. The left and right legs are identities and identities are isomorphisms, hence they are available in any stable system of monics.

```
idSpan : {A : Obj} → Span A A
idSpan {A} = record{
  A' = A;
  mhom = iden;
  fhom = iden;
  m∈sys = iso∈sys idIso}
```

The identity maps in the partial map category are given by `box idSpan`.

Let `ng : Span B C` and `mf : Span A B` be two spans. Let `n` and `m` be the left legs of `ng` and `mf` respectively, `g` and `f` be the right legs, and `n∈` and `m∈` be the proofs that `n` and `m` are in the stable system of monics. In order to form the composite span `compSpan ng mf`, we take the pullback of `f` along `n`. `W` is the object, and `h` and `k` are the two maps that complete the square underlying such pullback. The composite span is given by composing `h` with `m` and `k` with `g`. The span is well defined, since both `h` and `m` are in the stable system of monics, which is closed under composition.

```
compSpan : ∀{A B C} → Span B C → Span A B → Span A C
compSpan ng mf =
  let sq' = sq (proj₁ (pul∈sys (fhom mf) (m∈sys ng)))
  in record{
  A' = W sq';
  mhom = comp (mhom mf) (h sq');
  fhom = comp (fhom ng) (k sq');
  m∈sys =
    comp∈sys (proj₂ (pul∈sys (fhom mf) (m∈sys ng)))
             (m∈sys mf)}
```



We need a lemma ∼cong stating that _∼Span∼_ is a congruence with respect to composition of spans. compSpan is an operation on spans, so when reasoning about spans up to equality, we need that compSpan respects it.

```
∼cong : ∀{A B C}{ng n'g' : Span B C}{mf m'f' : Span A B}
  → mf ∼Span∼ m'f' → ng ∼Span∼ n'g'
  → compSpan ng mf ∼Span∼ compSpan n'g' m'f'
∼cong p r = ?
```

Composition is obtained by lifting the map λ x y → box (compSpan x y), which is compatible with _∼Span∼_. The eliminator $\texttt{lift}_2$ is the two-argument variant of lift.

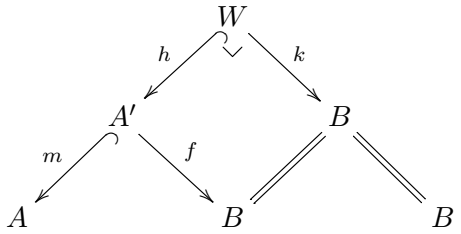
```
qcompSpan : ∀{A B C} → QSpan B C → QSpan A B → QSpan A C
qcompSpan =
  lift₂ _ (λ x y → box (compSpan x y))
        (λ p q → sound (∼cong p q))
```

The function qcompSpan is propositionally equal to box (compSpan ng mf), when applied to terms box ng and box mf. The computation rule $\texttt{lift}\beta_2$ is the two-argument variant of liftβ.

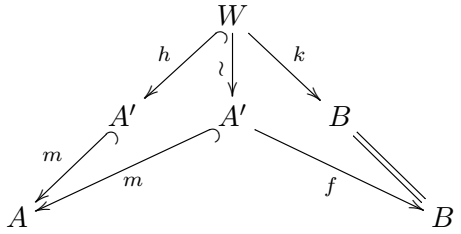
```
liftβComp : ∀{A B C}{ng : Span B C}{mf : Span A B}
  → qcompSpan (box ng) (box mf) ≡ box (compSpan ng mf)
liftβComp =
  liftβ₂ _ (λ x y → box (compSpan x y))
    (λ p q → sound (∼cong p q)) _ _
```

Next we present the proof of the left identity law for a partial map category. We have to prove that a span composed with the identity span is 'equal' to itself (up to _∼Span∼_). Let mf : Span A B be a span. Let A' be the object, m and f the left and right legs of mf. We take the pullback of the identity map along f given by the fact that the identity map is in every stable system of monics. We call W the object, and h and k the two maps that complete the square underlying such pullback. Let scom be the proof

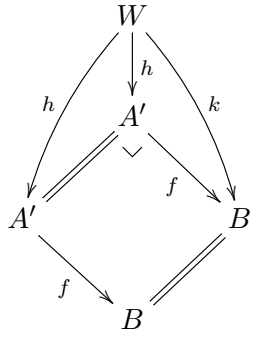
that the square commutes.



We have to find an isomorphism between `W` and `A'` that makes the two generated diagrams commute.



Note that there is also another pullback of the identity map along `f`, namely `trivialPullback f`. By the definition of `trivialPullback f`, the underlying morphism of the unique map to it from the given pullback is `h`; by `pullbackiso`, it is an isomorphism We supply `h` and the isomorphism proof together with the proofs that the two triangles in the diagram above commute. The first one is just `refl` and the second one follows immediately from the proof `scom` of the pullback of the identity map along `f`.




```
idlSpan : ∀{A B}{mf : Span A B}
  → compSpan idSpan mf ∼Span∼ mf
idlSpan {mf = mf} = record{
  let p = proj₁ (pul∈sys (fhom mf) (iso∈sys idIso))
      sq' = sq p
  in record{
```

```
  s = h sq';
  sIso = pullbackIso (trivialPullback (fhom mf)) p;
  leftTr∼ = refl;
  rightTr∼ = scom sq'}
```

We will use the quotient machinery in conjunction with this proof in the definition of the partial map category.

The composition `qcompSpan` is defined using $\texttt{lift}_2$. Therefore, when it is applied to arguments `box ng` and `box mf`, it is propositionally equal to `box (compSpan ng mf)`. Using this result, we prove the left identity law for the partial map category.

```
qidlSpan : ∀{A B}{x : QSpan A B}
  → qcompSpan (box idSpan) x ≡ x
qidlSpan {x = x} =
  lift (λ mf →
          proof
          qcompSpan (box idSpan) (box mf)
          ≡⟨ liftβComp ⟩
          box (compSpan idSpan mf)
          ≡⟨ sound idlSpan ⟩
          box mf
          ■)
         (fixtypes ∘ sound)
         x
```

In the above proof, we used the lemma `fixtypes`, useful to deal with the common situation where we have proofs of two equations with equal right hand sides.

```
fixtypes : {A B C D : Set}{a : A}{b : B}{c : C}{d : D}
  {p : a ≡ b}{q : c ≡ d} → b ≡ d → p ≡ q
fixtypes {p = refl}{refl} refl = refl
```

The right identity law and associativity of `qcompSpan` are proved similarly to the left identity law.

```
Par : Cat
Par = record{
  Obj  = Obj;
  Hom  = QSpan;
  iden = box idSpan;
  comp = qcompSpan;
  idl  = qidlSpan;
  idr  = ?;
  ass  = ?}
```

A.5 Restriction categories

A restriction category is a category with a restriction operation, i.e., every map comes with an endomap on its domain subject to four laws.

```
record RestCat : Set₁ where
  field
    cat  : Cat
    rest : ∀{A B} → Hom cat A B → Hom cat A A
    R1   : ∀{A B}{f : Hom cat A B}
      → comp cat f (rest f) ≡ f
    R2   : ∀{A B C}{f : Hom cat A B}{g : Hom cat A C}
      → comp cat (rest f) (rest g) ≡
        comp cat (rest g) (rest f)
    R3   : ∀{A B C}{f : Hom cat A B}{g : Hom cat A C}
      → comp cat (rest g) (rest f) ≡
        rest (comp cat g (rest f))
    R4   : ∀{A B C}{f : Hom cat A B}{g : Hom cat B C}
      → comp cat (rest g) f ≡
        comp cat f (rest (comp cat g f))
```

A restriction functor between two restriction categories is a functor between the underlying categories preserving the restriction operation.

```
record RestFun (C D : RestCat) : Set where
  field
    fun   : Fun (cat C) (cat D)
    frest : ∀{A B}{f : Hom (cat C) A B} →
            rest D (HMap fun f) ≡ HMap fun (rest C f)
```

The identity functor is always a restriction functor.

```
idRestFun : {C : RestCat} → RestFun C C
idRestFun = record{
  fun = idFun;
  frest = refl}
```

We fix a restriction category `X` with underlying category `Xcat`. We prove lemmata `lem1`, `lem3`, `lem2` and `lem4` (Lemma 4.1(i)-(iv) in Section 4.1.2). We show them here, since they are nice examples of the kind of equational reasoning one can do with restriction categories.

```
lem1 : ∀{A B}{f : Hom A B} → Mono f → rest f ≡ iden
lem1 {f = f} p =
  p (proof
```

```
      comp f (rest f)
      ≡⟨ R1 ⟩
      f
      ≡⟨ sym idr ⟩
      comp f iden
      ■)

lem2 : ∀{A B}{f : Hom A B}
  → comp (rest f) (rest f) ≡ rest f
lem2 {f = f} =
  proof
  comp (rest f) (rest f)
  ≡⟨ R3 ⟩
  rest (comp f (rest f))
  ≡⟨ cong rest R1 ⟩
  rest f
  ■

lem3 : ∀{A B}{f : Hom A B} → rest (rest f) ≡ rest f
lem3 {f = f} =
  proof
  rest (rest f)
  ≡⟨ cong rest (sym idl) ⟩
  rest (comp iden (rest f))
  ≡⟨ sym R3 ⟩
  comp (rest iden) (rest f)
  ≡⟨ cong (λ g → comp g (rest f)) (lem1 idMono) ⟩
  comp iden (rest f)
  ≡⟨ idl ⟩
  rest f
  ■

lem4 : ∀{A B C}{f : Hom A B}{g : Hom B C} →
        rest (comp g f) ≡ rest (comp (rest g) f)
lem4 {f = f}{g} =
  proof
  rest (comp g f)
  ≡⟨ cong (λ f' → rest (comp g f')) (sym R1) ⟩
  rest (comp g (comp f (rest f)))
  ≡⟨ cong rest (sym ass) ⟩
  rest (comp (comp g f) (rest f))
  ≡⟨ sym R3 ⟩
  comp (rest (comp g f)) (rest f)
```

```
  ≡⟨ R2 ⟩
  comp (rest f) (rest (comp g f))
  ≡⟨ R3 ⟩
  rest (comp f (rest (comp g f)))
  ≡⟨ cong rest (sym R4) ⟩
  rest (comp (rest g) f)
  ■
```

Notice how equational proofs in Agda look literally like those one would write by hand. E.g., compare the formal proof of `lem4` given above with the following pen-and-paper proof:

$$\overline{g \circ f} = \overline{g \circ (f \circ \overline{f})} = \overline{(g \circ f) \circ \overline{f}} = \overline{g \circ f} \circ \overline{f} = \overline{f} \circ \overline{g \circ f} = \overline{f \circ \overline{g \circ f}} = \overline{\overline{g} \circ f}$$

Restriction categories allow us to work with partial maps in a total setting. However, we still need to be able to identify total maps. In a restriction category, a total map is a map whose restriction is the identity map.

```
record Tot (A B : Obj) : Set where
  field
    hom     : Hom A B
    totProp : rest hom ≡ iden {A}
```

We need a lemma `totEq` stating that two total maps are equal, if their underlying morphisms are equal. This is a consequence of uniqueness of identity proofs.

```
totEq : ∀{A B}{f g : Tot A B} → hom f ≡ hom g → f ≡ g
totEq p = ?
```

The category `Total` of total maps in `X` inherits its identity `idTot` and composition `compTot` from the underlying category `Xcat`, but we must prove that the totality property `totProp` is satisfied. For the identity map `idTot`, the condition follows from the facts that identity maps are monic (`idMono`) and monic maps are total (`lem1`).

```
idTot : ∀{A} → Tot A A
idTot = record{
  hom = iden;
  totProp = lem1 idMono}
```

Given two total maps `g` and `f`, the totality condition `compTotProp` for the composite `compTot g f` follows from totality of `g` and `f` and `lem4`.

```
compTotProp : ∀{A B C}{g : Tot B C}{f : Tot A B}
  → rest (comp (hom g) (hom f)) ≡ iden
compTotProp {g = g}{f} =
  proof
  rest (comp (hom g) (hom f))
  ≡⟨ lem4 ⟩
  rest (comp (rest (hom g)) (hom f))
  ≡⟨ cong (λ h → rest (comp h (hom f))) (totProp g) ⟩
  rest (comp iden (hom f))
  ≡⟨ cong rest idl ⟩
  rest (hom f)
  ≡⟨ totProp f ⟩
  iden
  ■

compTot : ∀{A B C}(g : Tot B C)(f : Tot A B) → Tot A C
compTot g f = record{
  hom = comp (hom g) (hom f);
  totProp = compTotProp}
```

Having defined identities and composition, we can now define the category of total maps. The `totEq` lemma reduces the laws of a category to those of the underlying category.

```
Total : Cat
Total = record{
  Obj  = Obj;
  Hom  = Tot;
  iden = idTot;
  comp = compTot;
  idl  = totEq idl;
  idr  = totEq idr;
  ass  = totEq ass}
```

A.6 Soundness

The next step in the formalization is the soundness theorem, which states that any partial map category is a restriction category. In order to prove it, we equip the given partial map category with a restriction category structure (a restriction operator, proofs of R1, R2, R3 and R4). We perform the construction in two steps: first on spans and then we port it to quotiented spans, as we did in the definition of partial map categories. We fix a category `X` and a stable system of monics `M`. The restriction on spans simply copies the left leg of a span into the right leg position.

```
restSpan : ∀{A B} → Span A B → Span A A
restSpan mf = record{
  A' = A' mf;
  mhom = mhom mf;
  fhom = mhom mf;
  m∈sys = m∈sys mf}
```

We require that restriction respects the equivalence relation on spans. This is easy: the left commuting triangle `q` is copied to the right.

```
∼congRestSpan : ∀{A B}{mf ng : Span A B} → mf ∼Span∼ ng
  → restSpan mf ∼Span∼ restSpan ng
∼congRestSpan eq = record{
  s = s eq;
  sIso = sIso eq;
  leftTr∼ = leftTr∼ eq;
  rightTr∼ = leftTr∼ eq}
```

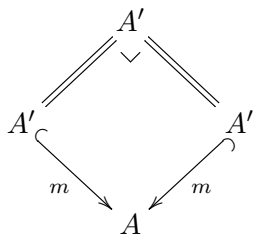
We port the restriction operator on spans to quotiented spans. We post-compose `restSpan` with `box`, obtaining a map from `Span A B` to `QSpan A A`. Then we lift this map. Compatibility follows from axiom `sound` and the above proved congruence `∼congRestSpan`.

```
qrestSpan : ∀{A B} → QSpan A B → QSpan A A
qrestSpan = lift (box ∘ restSpan) (sound ∘ ∼congRestSpan)
```

The function `qrestSpan` is propositionally equal to `box (restSpan mf)`, when applied to a term `box mf`.

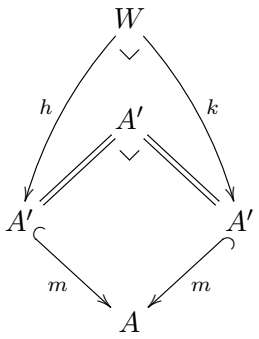
```
liftβRest : ∀{A B}{mf : Span A B}
  → qrestSpan (box mf) ≡ box (restSpan mf)
liftβRest =
  liftβ _ (box ∘ restSpan) (sound ∘ ∼congRestSpan) _
```

To prove R1 for the partial map category, we will use the basic fact that one can construct the following pullback from any monic map.

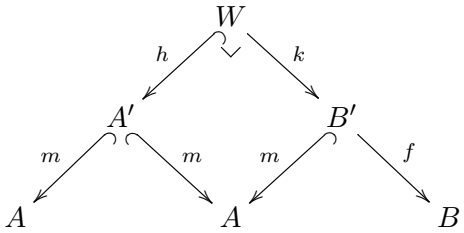



```
monicPullback : ∀{A' A}{m : Hom A' A} → Mono m → Pullback m m
monicPullback p = ?
```

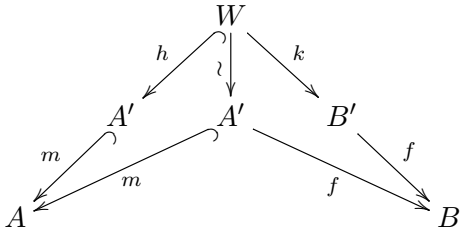
R1 states that composing a map `f` with its restriction is the same as `f`. We prove this property up to equivalence of spans first. Let `mf : Span A B` be a span. Let `A'` be the object, `m` and `f` the left and right legs of `mf`, and `m∈` the proof that `m` is in the stable system of monics. Note that there are two pullbacks of `m` along itself: (i) the pullback `monicPullback (mono∈sys m∈)`, where `mono∈sys m∈` is a proof that `m` is monic; (ii) the pullback given by the fact that `m` is in the stable system of monics, and therefore the pullback of `m` along any map exists. We call `W` the object, `h` and `k` the two maps that complete the square underlying the pullback (ii), and `scom` the proof that the square commutes.



We need to prove that the spans `compSpan mf (restSpan mf)` and `mf` are in the relation `∼Span∼`. Remember that the first span is constructed as follows:



Therefore, we have to find an isomorphism between `W` and `A'` that makes the two generated diagrams commute.



The map `h` does the job. The left diagram commutes by reflexivity. The right diagram commutes because `h ≡ k`, and this follows from `scom` and `m`

being a monic map. Note moreover that `h` is the unique map between the pullbacks (i) and (ii), and therefore it is an isomorphism.

```
R1Span : ∀{A B}{mf : Span A B}
  → compSpan mf (restSpan mf) ∼Span∼ mf
R1Span {mf = mf} =
  let p = proj₁ (pul∈sys (mhom mf) (m∈sys mf))
      sq' = sq p
  in record{
  s = h sq';
  sIso =
    pullbackIso (monicPullback (mono∈sys (m∈sys mf))) p;
  leftTr∼ = refl;
  rightTr∼ =
    cong (comp (fhom mf)) (mono∈sys (m∈sys mf) (scom sq'))}
```

The restriction `qrestSpan` is defined using `lift`, therefore, when applied to an argument `box mf`, it is propositionally equal to `box (restSpan mf)`. Having proved R1 up to `_∼Span∼_`, we can port this proof to `_≡_`.

```
qR1Span : ∀{A B}{x : QSpan A B}
  → qcompSpan x (qrestSpan x) ≡ x
qR1Span {x = x} =
  lift (λ x → qcompSpan x (qrestSpan x) ≡ x)
       (λ mf →
          proof
          qcompSpan (box mf) (qrestSpan (box mf))
          ≡⟨ cong (qcompSpan (box mf)) liftβRest ⟩
          qcompSpan (box mf) (box (restSpan mf))
          ≡⟨ liftβComp ⟩
          box (compSpan mf (restSpan mf))
          ≡⟨ sound R1Span ⟩
          box mf
          ■)
        (fixtypes ∘ sound)
        x
```

The proofs of the laws R2, R3 and R4 are performed in a similar way. This completes the proof of soundness (constructing a restriction category from a partial map category).

```
RestPar : RestCat
RestPar = record{
  cat = Par;
  rest = qrestSpan;
```

```
  R1 = qR1Span;
  R2 = ?;
  R3 = ?;
  R4 = ?}
```

A.7 Idempotents

We fix a category `X`. Idempotent maps in `X` are represented as records with three fields: an object `E`, an endomap `e` on `E` and a proof `idemLaw` of `comp e e ≅ e`. Our main use of idempotents will be as objects in a category so we choose to define them as below as opposed to as a predicate on maps (see `Mono`).

```
record Idem : Set where
  field
    E       : Obj
    e       : Hom E E
    idemLaw : comp e e ≅ e
```

The identity map on any object is an idempotent.

```
idIdem : {A : Obj} → Idem
idIdem {A} = record{
  E = A;
  e = iden;
  idemLaw = idl}
```

A class of idempotents `IdemClass` is given primarily in terms of a membership relation (see stable systems of monics `StableSys`). The second condition states that all identities are members.

```
record IdemClass : Set where
  field
    ∈class   : Idem → Set
    id∈class : ∀{A} → ∈class (idIdem {A})
```

A morphism between idempotents `i` and `i'` is a map between the underlying objects paired with a proof of an equation.

```
record IdemMor (i i' : Idem) : Set where
  field
    imap    : Hom (E i) (E i')
    imapLaw : comp (e i') (comp imap (e i)) ≅ imap
```

Two such morphisms are equal if their underlying maps are equal. This is a consequence of uniqueness of identity proofs.

```
idemMorEq : {i i' : Idem}{f g : IdemMor i i'}
  → imap f ≅ imap g → f ≅ g
idemMorEq p = ?
```

Every morphism `f : Hom A B` in the category `X` lifts to a morphism between idempotents `idIdem {A}` and `idIdem {B}`, since
`comp iden (comp f iden) ≅ f`.

```
idemMorLift : {A B : Obj}(f : Hom A B)
  → IdemMor (idIdem {A}) (idIdem {B})
idemMorLift f = record{
  imap = f;
  imapLaw =
    proof
    comp iden (comp f iden)
    ≅⟨ idl ⟩
    comp f iden
    ≅⟨ idr ⟩
    f
    ■}
```

In the proof of Lemma 2, we need the following property of a map `f` between idempotents `i` and `i'`: precomposing `imap f` with `e i` is equal to `imap f`. This is a direct consequence of the equality `imapLaw f`.

```
idemMorPrecomp : {i i' : Idem}{f : IdemMor i i'}
  → comp (imap f) (e i) ≅ imap f
idemMorPrecomp {i}{i'}{f} =
  proof
  comp (imap f) (e i)
  ≅⟨ cong (λ y → comp y (e i)) (sym (imapLaw f)) ⟩
  comp (comp (e i') (comp (imap f) (e i))) (e i)
  ≅⟨ cong (λ y → comp y (e i)) (sym ass) ⟩
  comp (comp (comp (e i') (imap f)) (e i)) (e i)
  ≅⟨ ass ⟩
  comp (comp (e i') (imap f)) (comp (e i) (e i))
  ≅⟨ cong (comp (comp (e i') (imap f))) (idemLaw i) ⟩
  comp (comp (e i') (imap f)) (e i)
  ≅⟨ ass ⟩
  comp (e i') (comp (imap f) (e i))
  ≅⟨ imapLaw f ⟩
  imap f
  ■
```

Analogously, one can prove that postcomposing `imap f` with `e i'` is equal to `imap f`.

```
idemMorPostcomp : {i i' : Idem}{f : IdemMor i i'}
  → comp (e i') (imap f) ≅ imap f
idemMorPostcomp {i}{i'} f =
  proof
  comp (e i') (imap f)
  ≅⟨ cong (comp (e i')) (sym (imapLaw f)) ⟩
  comp (e i') (comp (e i') (comp (imap f) (e i)))
  ≅⟨ sym ass ⟩
  comp (comp (e i') (e i')) (comp (imap f) (e i))
  ≅⟨ cong (λ y → comp y (comp (imap f) (e i))) (idemLaw i') ⟩
  comp (e i') (comp (imap f) (e i))
  ≅⟨ imapLaw f ⟩
  imap f
  ■
```

Idempotents and morphisms between them form a category. Identities are given by the idempotents themselves.

```
idIdemMor : {i : Idem} → IdemMor i i
idIdemMor {i} = record{
  imap = e i;
  imapLaw =
    proof
    comp (e i) (comp (e i) (e i))
    ≅⟨ cong (comp (e i)) (idemLaw i) ⟩
    comp (e i) (e i)
    ≅⟨ idemLaw i ⟩
    e i
    ■}
```

Composition is inherited from the underlying category.

```
compIdemMor : {i₁ i₂ i₃ : Idem}
  → (g : IdemMor i₂ i₃)(f : IdemMor i₁ i₂)
  → IdemMor i₁ i₃
compIdemMor {i₁}{i₂}{i₃} g f = record{
  imap = comp (imap g) (imap f);
  imapLaw =
    proof
    comp (e i₃) (comp (comp (imap g) (imap f)) (e i₁))
    ≅⟨ cong (comp (e i₃)) ass ⟩
    comp (e i₃) (comp (imap g) (comp (imap f) (e i₁)))
    ≅⟨ cong (λ y → comp (e i₃) (comp (imap g) y))
            idemMorPrecomp ⟩
```

```
    comp (e i₃) (comp (imap g) (imap f))
    ≅⟨ sym ass ⟩
    comp (comp (e i₃) (imap g)) (imap f)
    ≅⟨ cong (λ y → comp y (imap f)) idemMorPostcomp ⟩
    comp (imap g) (imap f)
    ■}
```

The associativity law follows directly from the associativity law of the underlying category. The identity laws do not follow directly, since identities in this new category are idempotents, but they are immediate consequences of `idemMorPrecomp` and `idemMorPostcomp`.

```
SplitCat : IdemClass → Cat
SplitCat E = record{
  Obj = Σ Idem (∈class E);
  Hom = λ ip jq → IdemMor (proj₁ ip) (proj₁ jq);
  iden = idIdemMor;
  comp = compIdemMor;
  idl = idemMorEq idemMorPostcomp;
  idr = idemMorEq idemMorPrecomp;
  ass = idemMorEq ass}
```

Given a class of idempotents `E`, our category `X` is a full subcategory of `SplitCat E`. We define the inclusion functor `InclSplitCat`. It sends an object `A` to its corresponding identity `idIdem {A}`, which belongs to `E` by definition of class of idempotents, and it sends a morphism `f` to its lifting `idemMorLift f`. The functor laws hold trivially.

```
InclSplitCat : (E : IdemClass) → Fun X (SplitCat E)
InclSplitCat E = record{
  OMap  = λ A → idIdem {A} , id∈class E;
  HMap  = idemMorLift;
  fid   = idemMorEq refl;
  fcomp = idemMorEq refl}
```

Since `InclSplitCat` is basically identity on morphisms, it is easy to show that it is a full and faithful functor.

```
FullInclSplitCat : {E : IdemClass} → Full (InclSplitCat E)
FullInclSplitCat {f = f} = imap f , idemMorEq refl

FaithfulInclSplitCat : {E : IdemClass}
  → Faithful (InclSplitCat E)
FaithfulInclSplitCat refl = refl
```

Moreover, the category `SplitCat E` is a restriction category, if the original category `X` is a restriction category. So let `X` be a restriction category and `Xcat` its underlying category. We describe formally the restriction operation on `SplitCat E`. Given a morphism `f : IdemMor i i'` in `SplitCat E`, the restriction of `f` has `comp (rest (imap f)) (e i)` as underlying morphism in `Xcat` (this corresponds to the 'hat' operation described in Lemma 2).

```
restIdemMor : {i i' : Idem} → IdemMor i i' → IdemMor i i
restIdemMor {i} f = record{
  imap = comp (rest (imap f)) (e i);
  imapLaw =
    proof
    comp (e i) (comp (comp (rest (imap f)) (e i)) (e i))
    ≅⟨ cong (comp (e i)) ass ⟩
    comp (e i) (comp (rest (imap f)) (comp (e i) (e i)))
    ≅⟨ cong (comp (e i) ∘ comp (rest (imap f))) (idemLaw i) ⟩
    comp (e i) (comp (rest (imap f)) (e i))
    ≅⟨ cong (comp (e i)) R4 ⟩
    comp (e i) (comp (e i) (rest (comp (imap f) (e i))))
    ≅⟨ sym ass ⟩
    comp (comp (e i) (e i)) (rest (comp (imap f) (e i)))
    ≅⟨ cong (λ y → comp y (rest (comp (imap f) (e i))))
            (idemLaw i) ⟩
    comp (e i) (rest (comp (imap f) (e i)))
    ≅⟨ sym R4 ⟩
    comp (rest (imap f)) (e i)
    ■}
```

The restriction category axioms are easily provable. For example, R1 is a direct consequence of `X` being a restriction category and the property `idemMorPrecomp`. Remember that two parallel morphisms in `SplitCat E` are equal, if their underlying maps in `Xcat` are equal (a property we named `idemMorEq`).

```
R1Split : {E : IdemClass}{ip jq : Σ Idem (∈class E)}
  → {f : IdemMor (proj₁ ip) (proj₁ jq)}
  → compIdemMor f (restIdemMor f) ≅ f
R1Split {ip = i , p}{f = f} =
  idemMorEq
    (proof
     comp (imap f) (comp (rest (imap f)) (e i))
     ≅⟨ sym ass ⟩
     comp (comp (imap f) (rest (imap f))) (e i)
     ≅⟨ cong (λ y → comp y (e i)) R1 ⟩
```

```
    comp (imap f) (e i)
    ≅⟨ idemMorPrecomp ⟩
    imap f
    ■)
```

Proofs for R2, R3 and R4 are performed in a similar way.

```
RestSplitCat : (E : IdemClass) → RestCat
RestSplitCat E = record{
  cat = SplitCat E;
  rest = restIdemMor;
  R1 = R1Split;
  R2 = ?;
  R3 = ?;
  R4 = ?}
```

Lemma 2 can now be proved. Any restriction category `X` embeds fully in the restriction category `RestSplitCat E` for any class of idempotents `E`.

```
InclRestSplitCat : (E : IdemClass)
  → RestFun X (RestSplitCat E)
InclRestSplitCat E = record{
  fun = InclSplitCat E;
  frest = idemMorEq idr}
```

A.8 Restriction idempotents

We fix a restriction category `X` with underlying category `Xcat`. We define a predicate `isRestIdem` stating that an idempotent is a restriction idempotent. An idempotent is a restriction idempotent, if it is equal to its restriction.

```
isRestIdem : Idem → Set
isRestIdem i = e i ≅ rest (e i)
```

Restriction idempotents define a class of idempotents `restIdemClass`. Identity maps belong to the class, since they are monic (`idMono`) and monic maps are total (`lem1`).

```
restIdemClass : IdemClass
restIdemClass = record{
  ∈class = isRestIdem;
  id∈class = sym (lem1 idMono)}
```

A splitting of an idempotent `i` on an object `E` is a record consisting of an object `B`, a section from `B` to `E`, a retraction from `E` to `B` and proofs of two equations.

```
record Split (i : Idem) : Set where
  field
    B         : Obj
    sec       : Hom B (E i)
    retr      : Hom (E i) B
    splitLaw1 : comp sec retr ≅ e i
    splitLaw2 : comp retr sec ≅ iden {B}
```

A restriction category where all restriction idempotents are split is called a split restriction category.

```
record SplitRestCat : Set where
  field
    rcat          : RestCat
    restIdemSplit : (i : Idem (cat rcat))
      → isRestIdem rcat i → Split (cat rcat) i
```

Lemma 3 states that the restriction category `RestSplitCat restIdemClass` (built from a restriction category `X`) is a split restriction category. For readability and simplicity reasons, we do not show the proof that every restriction idempotent is split.

```
SplitRestSplitCat : SplitRestCat
SplitRestSplitCat = record{
  rcat = RestSplitCat restIdemClass;
  restIdemSplit = ?}
```

A.9 Completeness

In order to state the completeness theorem (Theorem 4.2), we construct the stable system of monics `SectionsOfRestIdem` given by the sections of the restriction idempotents of a particular split restriction category. We fix a split restriction category `X` with underlying restriction category `Xrcat` and underlying category `Xcat`. First, we define a record `SectionOfRestIdem` parametrized by a total map `s`. The proposition `SectionOfRestIdem s` holds if and only if `hom s` is a section of a restriction idempotent.

```
record SectionOfRestIdem {B E} (s : Tot B E) : Set where
  field
    e         : Hom E E
    restIdem  : e ≅ rest e
    r         : Hom E B
    splitLaw1 : comp (hom s) r ≅ e
    splitLaw2 : comp r (hom s) ≅ iden {B}
```

The predicate `SectionOfRestIdem` defines a stable system of monics in the subcategory of total maps in `rcat`. We show that every isomorphism is a section of a restriction idempotent. We do not show the proof that every map in the system is monic and the proofs that the system is closed under composition and pullback. Note that identity maps are restriction idempotents and every isomorphism is the section of an identity map.

```
iso∈sysSectionOfRestIdem : ∀{B E}{s : Tot B E} → Iso s
  → SectionOfRestIdem s
iso∈sysSectionOfRestIdem i = record{
  e = iden;
  restIdem = sym (lem1 (idMono cat));
  r = hom (inv i);
  splitLaw1 = cong hom (rinv i);
  splitLaw2 = cong hom (linv i)}

SectionOfRestIdemSys : StableSys Total
SectionOfRestIdemSys = record{
  ∈sys = SectionOfRestIdem;
  mono∈sys = ?;
  iso∈sys = iso∈sysSectionOfRestIdem;
  comp∈sys = ?;
  pul∈sys = ?}
```

We now move to the formalization of the completeness theorem (Theorem 4.2) for a particular split restriction category `X` with underlying restriction category `Xrcat` and underlying category `Xcat`. Let `Par` be the partial map category over the category of total maps `Total` and stable system of monics `SectionsOfRestIdemSys`, and `RestPar` the restriction category on top of `Par` given by soundness. We show the construction of the functors `Funct : Fun Xcat Par` and `Funct2 : Fun Par Xcat`. These functors can be lifted to restriction functors `RFunct` and `RFunct2` and they are each other inverses in the category of restriction categories and restriction functors, therefore showing that `Xrcat` and `RestPar` are isomorphic in this category.

The functor `Funct` is identity on objects. The mapping of maps of the functor `Funct` takes a map `f : Hom A C` in `Xcat` and returns a map in `Par`, i.e., an element of `QSpan A C`. We first define a function `HMap1` that constructs a span between `A` and `C`. The mapping of maps of `Funct` will be `abs ∘ HMap1`. The map `rest f` is a an idempotent (`lem2`), moreover a restriction idempotent (`lem3`), therefore it splits.

```
restIdemIdemGen : ∀{A C}(f : Hom A C) → Idem
restIdemIdemGen {A} f = record{
  E = A;
```

```
  e = rest f;
  idemLaw = lem2}

restIdemSplitGen : ∀{A C}(f : Hom A C)
  → Split (restIdemIdemGen f)
restIdemSplitGen f =
  restIdemSplit (restIdemIdemGen f) (sym lem3)
```

For the left leg of the span, we take the section `sec (restIdemSplitGen f)`, which is total. For the right leg, we take the map `comp f (sec (restIdemSplitGen f))`, which is also total.

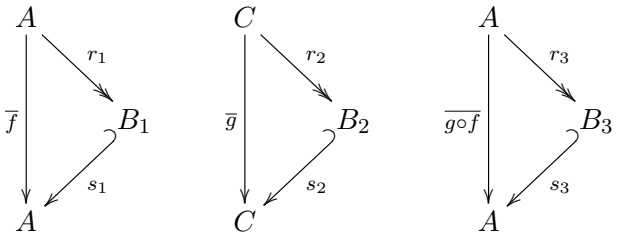
```
leftLeg : ∀{A C}(f : Hom A C)
  → Tot (B (restIdemSplitGen f)) A
leftLeg f = record{
  hom = sec (restIdemSplitGen f);
  totProp = ?}

rightLeg : ∀{A C}(f : Hom A C)
  → Tot (B (restIdemSplitGen f)) C
rightLeg f = record{
  hom = comp f (sec (restIdemSplitGen f));
  totProp = ?}
```

This concludes the definition of the functor `Funct`. The total map `leftLeg f` is the section of a restriction idempotent, i.e., there exists an element of type `SectionOfRestIdem (leftLeg f)`.

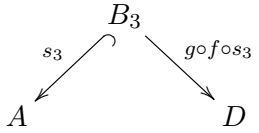
```
HMap1 : ∀{A C}(f : Hom A C) → Span A C
HMap1 f = record{
  A' = B (restIdemSplitGen f);
  mhom = leftLeg f;
  fhom = rightLeg f;
  m∈sys = ?}
```

`HMap1` is required to preserve identities and composition. Here we show that it preserves composition up to `_∼Span∼_`. Let `f : Hom A C` and `g : Hom C D`. We prove that the span `HMap1 (comp g f)` is related to `compSpan (HMap1 g) (HMap1 f)` by `_∼Span∼_`. Since the restriction idem-

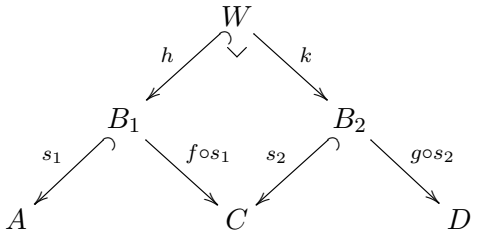
potents split, in particular we have the three following diagrams.



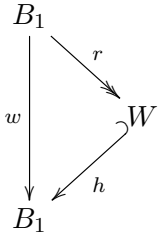
The span `HMap1 (comp g f)` is



while the span `compSpan (HMap1 g) (HMap1 f)` is



Notice that the map `h` is in the stable system of monics, i.e. it is the section of a restriction idempotent. This is true because `h` is the pullback of $\mathtt{s}_2$, which is in the stable system of monics. In particular, there exists a restriction idempotent `w : Hom W W` and a map `r : Hom` $\mathtt{B}_1$ `W` such that the following triangle commutes.



Our goal is to find an isomorphism `u : Hom` $\mathtt{B}_3$ `W` that makes the two generated triangles commute. It is not difficult to show that the composite map

$$u = B_3 \xrightarrow{s_3} A \xrightarrow{r_1} B_1 \xrightarrow{r} W$$

does the job. As usual, we refer to the Agda formalization for more details.

We obtain a functor `Funct` between the categories `Xcat` and `Par`.

```
Funct : Fun Xcat Par
Funct = record{
  OMap = id;
  HMap = abs ∘ HMap1;
  fid = ?;
  fcomp = ?}
```

The functor `Funct` also preserves the restriction operation. Therefore it is a restriction functor.

```
RFunct : RestFun Xrcat RestPar
RFunct = record{
  fun = Funct;
  frest = ?}
```

The functor `Funct2` is also identity on objects. The mapping of maps of the functor `Funct2` takes an element of `QSpan A C` into a map between `A` and `C`. We first define a function `HMap2` from `Span A C` into `Hom A C`. We fix a span `mf`. Let `A'` the object, `m` and `f` be the left and right legs of `mf` (which are total maps), and `m∈` be the proof that `m` is in the stable system of monics, i.e., `m∈` states that the total map `m` is the section of a restriction idempotent. The morphisms `hom f : Hom A' C` and `r m∈ : Hom A A'` are composable, and their composition defines `HMap2`.

```
HMap2 : ∀{A C} → Span A C → Hom A C
HMap2 mf = comp (hom (fhom mf)) (r (m∈sys mf))
```

`HMap2` is compatible with the equivalence relation `∼Span∼` on `Span A C`. So it can be lifted to a function `qHMap2` on the quotient `QSpan A C`. This concludes the description of the functor `Funct2`.

```
qHMap2 : ∀{A C} → QSpan A C → Hom A C
qHMap2 {A}{C} = lift {A}{C} HMap2 ?
```

The function `qHMap2` is propositionally equal to `HMap2 mf`, when applied to a term `abs nm`.

```
liftβqHMap2 : ∀{A C}{mf : Span A C}
  → qHMap2 (abs mf) ≅ HMap2 mf
liftβqHMap2 = liftβ _ HMap2 ? _
```

It is not difficult to see that `qHMap2` preserves identities and composition. We obtain a functor `Funct2` between `Par` and `Xcat`.

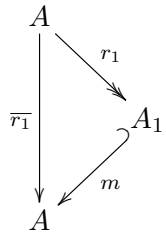
```
Funct2 : Fun Par Xcat
Funct2 = record{
  OMap = id;
```

```
  HMap = qHMap2;
  fid = ?;
  fcomp = ?}
```

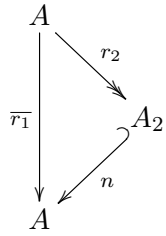
The functor `Funct2` preserves the restriction operation. Therefore it is a restriction functor.

```
RFunct2 : RestFun RestPar Xrcat
RFunct2 = record{
  fun = Funct2;
  frest = ?}
```

The functors `RFunct` and `RFunct2` are each other inverses. First, consider `mf : Span A C`. We show that `HMap1 (HMap2 mf) ~Span~ mf`. Let `m : Hom A₁ A` be the left leg of `mf` and `f : Hom A₁ C` the right leg. The map `m` is the section of a restriction idempotent. It is possible to prove that it is the section of `rest r₁`, where `r₁` is the retraction of the splitting. In particular, the following diagram commutes.



Let `n : Hom A₂ A` be the left leg of `HMap1 (HMap2 mf)`, the right leg is `comp (comp f r₁) n` by construction. The map `n` is the section of the restriction idempotent `rest (comp f r₁)`, and the latter is equal to `rest r₁` because `f` is total. In particular, there exists a map `r₂ : Hom A A₂` making the following diagram commute.



It is not difficult to prove that the map `comp r₁ n : Hom A₂ A₁` is an isomorphism between the spans `HMap1 (HMap2 mf)` and `mf`. This construction lifts straightforwardly to the quotient `QSpan A C`.

```
HIso1 : ∀{A C}(mf : QSpan A C) → abs (HMap1 (qHMap2 mf)) ≅ mf
HIso1 mf = ?
```

On the other hand, consider a map `f : Hom A C`. The map `HMap2 (HMap1 f)` is given by `comp (comp f s) r`, where `s` and `r` are the section and the retraction of the splitting of `rest f`.

```
HIso2 : ∀{A C}(f : Hom A C) → qHMap2 (abs (HMap1 f)) ≅ f
HIso2 f =
  let open Split (restIdemSplitGen f)
  in
    proof
    qHMap2 (abs (HMap1 f))
    ≅⟨ qHMap2liftβ ⟩
    comp (comp f s) r
    ≅⟨ ass ⟩
    comp f (comp s r)
    ≅⟨ cong (comp f) splitLaw1 ⟩
    comp f (rest f)
    ≅⟨ R1 ⟩
    f
    ■
```

This completes the proof of Theorem 4.2: every split restriction category is isomorphic to a partial map category in the category of restriction categories and restriction functors.

Acknowledgements

My sincere gratitude goes to my supervisors Tarmo Uustalu and James Chapman. They have been a great source of inspiration for me. My personal approach towards research was heavily shaped by our numerous discussions. Their enthusiasm has always been a strong driving power.

Big thanks go to my girlfriend Maria, especially for her extraordinary patience and optimism. She single-handedly revolutionized my life in Estonia, and this had a huge impact on my work and on this thesis in particular.

Very special thanks go to my daughter Anita. Her curiosity made me more curious, her joyfulness made me more joyful and her getting up at an ungodly hour made my days longer and more productive.

Huge thanks go to my parents Barbara and Nicola, for their unconditional support and encouragement.

Many thanks also go to my colleagues at the Institute of Cybernetics, in particular the other members of the Logic and Semantics group: Silvio, Wolfgang and Hendrik. Special thanks go to Denis for the many discussions and suggestions.

I would also like to thank Martin Escardó for suggesting the idea of constructing the quotiented delay monad using the axiom of propositional choice.

My research was supported by the ERDF funded national centre of excellence EXCS (3.2.0101.08-0013) and national ICT programme project Coinduction (3.2.1201.13-0029), the Estonian Ministry of Education and Research institutional research grant no. IUT33-13 and the Estonian Research Council personal research grant no. PUT763. In addition, I received support from the Estonian Science Foundation project grant no. 9398, the ESF funded ICT doctoral school (1.2.0401.09-0081) and the Estonian IT Academy programme.

Abstract

In this thesis, we continue the study of Capretta's coinductive delay monad in Martin-Löf type theory. The delay monad constitutes a viable constructive alternative to the maybe monad and allows the implementation of possibly non-terminating computations. Its applications range from the representation of general recursive functions to the formalization of domain theory, from the operational semantics for While languages to normalization by evaluation.

In all these applications, one is only interested in the terminating/non-terminating behavior of a computation, and not in its rate of convergence. This is equivalent to working with the delay datatype quotiented by weak bisimilarity. Our first main result is the discovery that the delay datatype quotiented by weak bisimilarity does not inherit the monad structure immediately. This has to do with the coinductive nature of the delay datatype and the bad interaction between inductive-like quotients in the style of Hofmann and infinitary types such as non-wellfounded trees. In order to construct a monad structure on the delay type, we need to postulate additional classical or semi-classical principles, such as the limited principle of omniscience or a certain weak version of the axiom of countable choice. These principles are also necessary for proving that the quotiented delay monad delivers free ω-complete pointed partial orders.

We can say that the quotiented delay monad is an useful tool for modeling partiality as an effect in type theory. Our second main result is to make the latter statement rigorous. We introduce a class of monads for encoding non-termination as an effect. A monad in this class is named a ω-complete pointed classifying monad, which formally is a monad whose Kleisli category is a restriction category à la Cockett and Lack, which moreover is $\omega\mathbf{CPPO}$-enriched with respect to the restriction order and in which pure maps are total. We show that the quotiented delay monad is the initial ω-complete pointed classifying monad in type theory. This universal property singles it out from among other examples of such monads, for examples from other partial map classifiers specified in terms of countably-complete dominances.

From a more general point of view, we ask ourselves whether type-theoretical approaches to partiality could possibly benefit from category-theoretical ones. Although we do not have a complete answer to this ques-

tion yet, we present the first steps in this direction. Our last main result consists of an Agda formalization of the first chapters of the theory of Cockett and Lack's restriction categories. Notably, it includes the proof of their completeness with respect to partial map categories, the latter being the standard generalization of sets and partial functions to more general categories. We hope that our development can become the cornerstone of a flexible framework for partiality in dependently typed programming languages, allowing one to program and reason about partial functions on different levels of abstraction.

Resümee

Käesolevas doktoritöös uurime Capretta koinduktiivset hilistusmonaadi Martin-Löfi tüübiteoorias. Hilistusmonaad on optsioonimonaadi konstruktiivne alternatiiv, mis lubab tüübiteoorias väljendada mittetermineeruvaid arvutusi. Hilistusmonaadi rakendused ulatuvad üldrekursiivsete funktsioonide esitamisest domeeniteooria formaliseerimiseni, While'i laadsete keete operatsioonsemantikast normaliseerimiseni väärtustamise kaudu.

Kõigile neile rakendustele on iseloomulik, et huvi tuntakse ainult arvutuse termineeruvuse või mittetermineeruvuse vastu, termineeruvuse kiirus ei ole oluline. See vastab hilistusmonaadi faktoriseerimisele nõrga bisimilaarsuse suhtes. Meie esimeseks tulemuseks on avastus, et nõrga bisimilaarsuse suhtes faktoriseeritud hilistusfunktor ei päri vahetult selle monaadilist struktuuri. See on tingitud hilistusfunktori koinduktiivsest loomust ja asjaolust, et Hofmanni stiilis faktortüübid ning infinitaarsed andmetüübid nagu fundeerimata puude tüübid ei tööta koos hästi. Selleks, et faktoriseeritud hilistusfunktoril rekonstrueerida monaadi struktuur, peame postuleerima klassikalisi või poolklassikalisi printsiipe nagu piiratud kõigeteadmise printsiip või nõrk loenduva valiku aksioom. Ilma neid printsiipe eeldamata ei saa ka tõestada, et faktoriseeritud hilistusmonaad tagastab vabu ω-täielikke punkteeritud osalisi järjestusi.

Hilistusmonaad on niisiis sobiv tööriist mittetermineeruvuse või osalisuse tüübiteoreetiliseks modelleerimiseks efektina. Meie teine tulemus annab sellele sobilikkusele täpse karakterisatsiooni. Me toome sisse klassi monaade mittetermineeruvuse kodeerimiseks efektina, nn. ω-täielikud punkteeritud klassifitseerivad monaadid. Need on monaadid, mille Kleisli kategooria on Cocketti ja Lacki mõttes kitsenduste kategooria, mis on ω**CPPO**-rikastatud kitsendusjärjestuse suhtes ja milles puhtad morfismid on totaalsed. Me näitame, et faktoriseeritud hilistusmonaad on initsiaalne ω-täielik punkteeritud klassifitseeriv monaad tüübiteoorias. See universaalomadus eristab teda muudest sarnastest monaadidest, nt teistest loenduvalt täielike domineeringute kaudu defineeritavatest osaliste morfismide klassifikaatoritest.

Võib küsida, millist praktilist kasu võivad mittetermineeruvuse kategooriateoreetilised käsitlused tüübiteooria jaoks pakkuda. Kuigi meil pole sellele küsimusele täielikku vastust, esitleme esimesi samme selle suunas. Meie viimaseks tulemuseks on Cocketti ja Lacki kitsenduste kategooriate teooria

esimeste peatükkide formalisatsioon interaktiivse tõestusassistendiga Agda. Olulise komponendina sisaldab see arendus kitsenduste kategooriate täielikkuse teoreemi osaliste morfismide kategooriate suhtes. Viimased kujutavad endast hulkade ja osaliste funktsioonide kategooria konstruktsiooni standardset üldistust üldistele kategooriatele. Me loodame, et sellest formalisatsioonist kasvab välja paindlik raamistik mittetermineeruvuse modelleerimiseks sõltuvalt tüübitud programmeerimiskeeltes, milles mittetermineeruvust saab programmeerida ja selle üle arutleda eri abstraktsioonitasemetel.

Curriculum Vitae

1. Personal data

Name	Niccolò Veltri
Date and place of birth	26 August 1988, Firenze, Italy
Citizenship	Italian
E-mail address	niccolo@cs.ioc.ee

2. Education

Educational institution	Period	Degree
Tallinn University of Technology	2012–2017	PhD studies
University of Florence	2010–2012	Master of Science
University of Florence	2007–2010	Bachelor of Science

3. Language skills

Language	Level
Italian	native
English	fluent

4. Special courses

Period	Event
5–10 Mar. 2017	22nd Estonian Winter School in Computer Science
28 Feb.–4 Mar. 2016	21st Estonian Winter School in Computer Science
15–27 June 2015	Oregon Programming Languages Summer School
1–6 Mar. 2015	20th Estonian Winter School in Computer Science
20–25 July 2014	"Proof, Truth, Computation" Summer School
2–7 Mar. 2014	19th Estonian Winter School in Computer Science
8–12 Apr. 2013	Midlands Graduate School 2013
3–8 Mar. 2013	18th Estonian Winter School in Computer Science

5. Professional employment

Period	Organisation	Position
2012–...	Inst. of Cybernetics at TUT	junior researcher

6. Research activity

   Type theory, constructive mathematics, formalization of mathematics, category theory.

7. Publications

1. Chapman, J., Uustalu, T., Veltri, N.: Quotienting the delay monad by weak bisimilarity. In: Leucker, M., Rueda, C., Valencia, F. D. (eds.) Proc. of 12th Int. Coll. on Theoretical Aspects of Computing, ICTAC 2015, Lect. Notes in Comput. Sci., v. 9399, pp. 110–125, Springer (2015)
2. Veltri, N.: Two set-based implementations of quotients in type theory. In: Nummenmaa, J., Sievi-Korte, O., Mäkinen, E. (eds.) Proc. of 14th Symposium on Programming Languages and Software Tools, SPLST 2015, CEUR Workshop Proceedings, v. 1525, pp. 194–205, CEUR-WS.org (2015)
3. Firsov, D., Uustalu, T., Veltri, N.: Variations on Noetherianness. In: Atkey, R., Krishnaswami, N. (eds.) Proc. of 6th Wksh. on Mathematically Structured Functional Programming, MSFP 2016, Electron. Proc. in Theor. Comput. Sci., v. 207, pp. 76–88, Open Publishing Assoc. (2016)
4. Chapman, J., Uustalu, T., Veltri, N.: Formalizing restriction categories. J. Formalized Reasoning, 10(1), 1–36 (2017)
5. Uustalu, T., Veltri, N.: Finiteness and rational sequences, constructively. J. Funct. Program., 27, article e13 (2017)

Elulookirjeldus

1. Isikuandmed

Ees- ja perekonnanimi	Niccolò Veltri
Sünniaeg ja -koht	26.08.1988, Firenze, Itaalia
Kodakondsus	Itaalia
E-posti aadress	niccolo@cs.ioc.ee

2. Hariduskäik

Õppeasutus	Õppimise aeg	Haridus
Tallinna Tehnikaülikool	2012–2017	Doktoriõpe
Firenze Ülikool	2010–2012	Magistrikraad
Firenze Ülikool	2007–2010	Bakalaureusekraad

3. Keelteoskus

Keel	Tase
Itaalia keel	emakeel
Inglise keel	kõrgtase

4. Täiendusõpe

Õppimise aeg	Täiendusõppe korraldaja nimetus
5.–10.03.17	22nd Estonian Winter School in Computer Science
28.02.–4.03.16	21st Estonian Winter School in Computer Science
15.–27.06-15	Oregon Programming Languages Summer School
1.–6.03.15	20th Estonian Winter School in Computer Science
20.–25.07.14	“Proof, Truth, Computation” Summer School
2.–7.03.14	19th Estonian Winter School in Computer Science
8.–12.04.13	Midlands Graduate School 2013
3.–8.03.13	18th Estonian Winter School in Computer Science

5. Teenistuskäik

Töötamise aeg	Tööandja nimetus	Ametikoht
2012–...	TTÜ Küberneetika Instituut	nooremteadur

6. Teadustegevus

   Tüübiteooria, konstruktiivne matemaatika, matemaatika formaliseerimine, kategooriateooria.

7. Publikatsioonid

   1. Chapman, J., Uustalu, T., Veltri, N.: Quotienting the delay monad by weak bisimilarity. In: Leucker, M., Rueda, C., Valencia, F. D. (eds.) Proc. of 12th Int. Coll. on Theoretical Aspects of Computing, ICTAC 2015, Lect. Notes in Comput. Sci., v. 9399, pp. 110–125, Springer (2015)
   2. Veltri, N.: Two set-based implementations of quotients in type theory. In: Nummenmaa, J., Sievi-Korte, O., Mäkinen, E. (eds.) Proc. of 14th Symposium on Programming Languages and Software Tools, SPLST 2015, CEUR Workshop Proceedings, v. 1525, pp. 194–205, CEUR-WS.org (2015)
   3. Firsov, D., Uustalu, T., Veltri, N.: Variations on Noetherianness. In: Atkey, R., Krishnaswami, N. (eds.) Proc. of 6th Wksh. on Mathematically Structured Functional Programming, MSFP 2016, Electron. Proc. in Theor. Comput. Sci., v. 207, pp. 76–88, Open Publishing Assoc. (2016)
   4. Chapman, J., Uustalu, T., Veltri, N.: Formalizing restriction categories. J. Formalized Reasoning, 10(1), 1–36 (2017)
   5. Uustalu, T., Veltri, N.: Finiteness and rational sequences, constructively. J. Funct. Program., 27, article e13 (2017)

DISSERTATIONS DEFENDED AT TALLINN UNIVERSITY OF TECHNOLOGY ON *INFORMATICS AND SYSTEM ENGINEERING*

1. **Lea Elmik**. Informational Modelling of a Communication Office. 1992.

2. **Kalle Tammemäe**. Control Intensive Digital System Synthesis. 1997.

3. **Eerik Lossmann**. Complex Signal Classification Algorithms, Based on the Third-Order Statistical Models. 1999.

4. **Kaido Kikkas**. Using the Internet in Rehabilitation of People with Mobility Impairments – Case Studies and Views from Estonia. 1999.

5. **Nazmun Nahar**. Global Electronic Commerce Process: Business-to-Business. 1999.

6. **Jevgeni Riipulk**. Microwave Radiometry for Medical Applications. 2000.

7. **Alar Kuusik**. Compact Smart Home Systems: Design and Verification of Cost Effective Hardware Solutions. 2001.

8. **Jaan Raik**. Hierarchical Test Generation for Digital Circuits Represented by Decision Diagrams. 2001.

9. **Andri Riid**. Transparent Fuzzy Systems: Model and Control. 2002.

10. **Marina Brik**. Investigation and Development of Test Generation Methods for Control Part of Digital Systems. 2002.

11. **Raul Land**. Synchronous Approximation and Processing of Sampled Data Signals. 2002.

12. **Ants Ronk**. An Extended Block-Adaptive Fourier Analyser for Analysis and Reproduction of Periodic Components of Band-Limited Discrete-Time Signals. 2002.

13. **Toivo Paavle**. System Level Modeling of the Phase Locked Loops: Behavioral Analysis and Parameterization. 2003.

14. **Irina Astrova**. On Integration of Object-Oriented Applications with Relational Databases. 2003.

15. **Kuldar Taveter**. A Multi-Perspective Methodology for Agent-Oriented Business Modelling and Simulation. 2004.

16. **Taivo Kangilaski**. Eesti Energia käiduhaldussüsteem. 2004.

17. **Artur Jutman**. Selected Issues of Modeling, Verification and Testing of Digital Systems. 2004.

18. **Ander Tenno**. Simulation and Estimation of Electro-Chemical Processes in Maintenance-Free Batteries with Fixed Electrolyte. 2004.

19. **Oleg Korolkov**. Formation of Diffusion Welded Al Contacts to Semiconductor Silicon. 2004.

20. **Risto Vaarandi**. Tools and Techniques for Event Log Analysis. 2005.

21. **Marko Koort**. Transmitter Power Control in Wireless Communication Systems. 2005.

22. **Raul Savimaa**. Modelling Emergent Behaviour of Organizations. Time-Aware, UML and Agent Based Approach. 2005.

23. **Raido Kurel**. Investigation of Electrical Characteristics of SiC Based Complementary JBS Structures. 2005.

24. **Rainer Taniloo**. Ökonoomsete negatiivse diferentsiaaltakistusega astmete ja elementide disainimine ja optimeerimine. 2005.

25. **Pauli Lallo**. Adaptive Secure Data Transmission Method for OSI Level I. 2005.

26. **Deniss Kumlander**. Some Practical Algorithms to Solve the Maximum Clique Problem. 2005.

27. **Tarmo Veskioja**. Stable Marriage Problem and College Admission. 2005.

28. **Elena Fomina**. Low Power Finite State Machine Synthesis. 2005.

29. **Eero Ivask**. Digital Test in WEB-Based Environment 2006.

30. **Виктор Войтович**. Разработка технологий выращивания из жидкой фазы эпитаксиальных структур арсенида галлия с высоковольтным p-n переходом и изготовления диодов на их основе. 2006.

31. **Tanel Alumäe**. Methods for Estonian Large Vocabulary Speech Recognition. 2006.

32. **Erki Eessaar**. Relational and Object-Relational Database Management Systems as Platforms for Managing Softwareengineering Artefacts. 2006.

33. **Rauno Gordon**. Modelling of Cardiac Dynamics and Intracardiac Bio-impedance. 2007.

34. **Madis Listak**. A Task-Oriented Design of a Biologically Inspired Underwater Robot. 2007.

35. **Elmet Orasson**. Hybrid Built-in Self-Test. Methods and Tools for Analysis and Optimization of BIST. 2007.

36. **Eduard Petlenkov**. Neural Networks Based Identification and Control of Nonlinear Systems: ANARX Model Based Approach. 2007.

37. **Toomas Kirt**. Concept Formation in Exploratory Data Analysis: Case Studies of Linguistic and Banking Data. 2007.

38. **Juhan-Peep Ernits**. Two State Space Reduction Techniques for Explicit State Model Checking. 2007.

39. **Innar Liiv**. Pattern Discovery Using Seriation and Matrix Reordering: A Unified View, Extensions and an Application to Inventory Management. 2008.

40. **Andrei Pokatilov**. Development of National Standard for Voltage Unit Based on Solid-State References. 2008.

41. **Karin Lindroos**. Mapping Social Structures by Formal Non-Linear Information Processing Methods: Case Studies of Estonian Islands Environments. 2008.

42. **Maksim Jenihhin**. Simulation-Based Hardware Verification with High-Level Decision Diagrams. 2008.

43. **Ando Saabas**. Logics for Low-Level Code and Proof-Preserving Program Transformations. 2008.

44. **Ilja Tšahhirov**. Security Protocols Analysis in the Computational Model – Dependency Flow Graphs-Based Approach. 2008.

45. **Toomas Ruuben**. Wideband Digital Beamforming in Sonar Systems. 2009.

46. **Sergei Devadze**. Fault Simulation of Digital Systems. 2009.

47. **Andrei Krivošei**. Model Based Method for Adaptive Decomposition of the Thoracic Bio-Impedance Variations into Cardiac and Respiratory Components. 2009.

48. **Vineeth Govind**. DfT-Based External Test and Diagnosis of Mesh-like Networks on Chips. 2009.

49. **Andres Kull**. Model-Based Testing of Reactive Systems. 2009.

50. **Ants Torim**. Formal Concepts in the Theory of Monotone Systems. 2009.

51. **Erika Matsak**. Discovering Logical Constructs from Estonian Children Language. 2009.

52. **Paul Annus**. Multichannel Bioimpedance Spectroscopy: Instrumentation Methods and Design Principles. 2009.

53. **Maris Tõnso**. Computer Algebra Tools for Modelling, Analysis and Synthesis for Nonlinear Control Systems. 2010.

54. **Aivo Jürgenson**. Efficient Semantics of Parallel and Serial Models of Attack Trees. 2010.

55. **Erkki Joasoon**. The Tactile Feedback Device for Multi-Touch User Interfaces. 2010.

56. **Jürgo-Sören Preden**. Enhancing Situation – Awareness Cognition and Reasoning of Ad-Hoc Network Agents. 2010.

57. **Pavel Grigorenko**. Higher-Order Attribute Semantics of Flat Languages. 2010.

58. **Anna Rannaste**. Hierarcical Test Pattern Generation and Untestability Identification Techniques for Synchronous Sequential Circuits. 2010.

59. **Sergei Strik**. Battery Charging and Full-Featured Battery Charger Integrated Circuit for Portable Applications. 2011.

60. **Rain Ottis**. A Systematic Approach to Offensive Volunteer Cyber Militia. 2011.

61. **Natalja Sleptšuk**. Investigation of the Intermediate Layer in the Metal-Silicon Carbide Contact Obtained by Diffusion Welding. 2011.

62. **Martin Jaanus**. The Interactive Learning Environment for Mobile Laboratories. 2011.

63. **Argo Kasemaa**. Analog Front End Components for Bio-Impedance Measurement: Current Source Design and Implementation. 2011.

64. **Kenneth Geers**. Strategic Cyber Security: Evaluating Nation-State Cyber Attack Mitigation Strategies. 2011.

65. **Riina Maigre**. Composition of Web Services on Large Service Models. 2011.

66. **Helena Kruus**. Optimization of Built-in Self-Test in Digital Systems. 2011.

67. **Gunnar Piho**. Archetypes Based Techniques for Development of Domains, Requirements and Sofware. 2011.

68. **Juri Gavšin**. Intrinsic Robot Safety Through Reversibility of Actions. 2011.

69. **Dmitri Mihhailov**. Hardware Implementation of Recursive Sorting Algorithms Using Tree-like Structures and HFSM Models. 2012.

70. **Anton Tšertov**. System Modeling for Processor-Centric Test Automation. 2012.

71. **Sergei Kostin**. Self-Diagnosis in Digital Systems. 2012.

72. **Mihkel Tagel**. System-Level Design of Timing-Sensitive Network-on-Chip Based Dependable Systems. 2012.

73. **Juri Belikov**. Polynomial Methods for Nonlinear Control Systems. 2012.

74. **Kristina Vassiljeva**. Restricted Connectivity Neural Networks based Identification for Control. 2012.

75. **Tarmo Robal**. Towards Adaptive Web – Analysing and Recommending Web Users` Behaviour. 2012.

76. **Anton Karputkin**. Formal Verification and Error Correction on High-Level Decision Diagrams. 2012.

77. **Vadim Kimlaychuk**. Simulations in Multi-Agent Communication System. 2012.

78. **Taavi Viilukas**. Constraints Solving Based Hierarchical Test Generation for Synchronous Sequential Circuits. 2012.

79. **Marko Kääramees**. A Symbolic Approach to Model-based Online Testing. 2012.

80. **Enar Reilent**. Whiteboard Architecture for the Multi-agent Sensor Systems. 2012.

81. **Jaan Ojarand**. Wideband Excitation Signals for Fast Impedance Spectroscopy of Biological Objects. 2012.

82. **Igor Aleksejev**. FPGA-based Embedded Virtual Instrumentation. 2013.

83. **Juri Mihhailov**. Accurate Flexible Current Measurement Method and its Realization in Power and Battery Management Integrated Circuits for Portable Applications. 2013.

84. **Tõnis Saar**. The Piezo-Electric Impedance Spectroscopy: Solutions and Applications. 2013.

85. **Ermo Täks**. An Automated Legal Content Capture and Visualisation Method. 2013.

86. **Uljana Reinsalu**. Fault Simulation and Code Coverage Analysis of RTL Designs Using High-Level Decision Diagrams. 2013.

87. **Anton Tšepurov**. Hardware Modeling for Design Verification and Debug. 2013.

88. **Ivo Müürsepp**. Robust Detectors for Cognitive Radio. 2013.

89. **Jaas Ježov**. Pressure sensitive lateral line for underwater robot. 2013.

90. **Vadim Kaparin**. Transformation of Nonlinear State Equations into Observer Form. 2013.

92. **Reeno Reeder**. Development and Optimisation of Modelling Methods and Algorithms for Terahertz Range Radiation Sources Based on Quantum Well Heterostructures. 2014.

93. **Ants Koel**. GaAs and SiC Semiconductor Materials Based Power Structures: Static and Dynamic Behavior Analysis. 2014.

94. **Jaan Übi**. Methods for Coopetition and Retention Analysis: An Application to University Management. 2014.

95. **Innokenti Sobolev**. Hyperspectral Data Processing and Interpretation in Remote Sensing Based on Laser-Induced Fluorescence Method. 2014.

96. **Jana Toompuu**. Investigation of the Specific Deep Levels in *p*-, *i*- and *n*-Regions of GaAs p^+-*pin*-n^+ Structures. 2014.

97. **Taavi Salumäe**. Flow-Sensitive Robotic Fish: From Concept to Experiments. 2015.

98. **Yar Muhammad**. A Parametric Framework for Modelling of Bioelectrical Signals. 2015.

99. **Ago Mõlder**. Image Processing Solutions for Precise Road Profile Measurement Systems. 2015.

100. **Kairit Sirts**. Non-Parametric Bayesian Models for Computational Morphology. 2015.

101. **Alina Gavrijaševa**. Coin Validation by Electromagnetic, Acoustic and Visual Features. 2015.

102. **Emiliano Pastorelli**. Analysis and 3D Visualisation of Microstructured Materials on Custom-Built Virtual Reality Environment. 2015.

103. **Asko Ristolainen**. Phantom Organs and their Applications in Robotic Surgery and Radiology Training. 2015.

104. **Aleksei Tepljakov**. Fractional-order Modeling and Control of Dynamic Systems. 2015.

105. **Ahti Lohk**. A System of Test Patterns to Check and Validate the Semantic Hierarchies of Wordnet-type Dictionaries. 2015.

106. **Hanno Hantson**. Mutation-Based Verification and Error Correction in High-Level Designs. 2015.

107. **Lin Li**. Statistical Methods for Ultrasound Image Segmentation. 2015.

108. **Aleksandr Lenin**. Reliable and Efficient Determination of the Likelihood of Rational Attacks. 2015.

109. **Maksim Gorev**. At-Speed Testing and Test Quality Evaluation for High-Performance Pipelined Systems. 2016.

110. **Mari-Anne Meister**. Electromagnetic Environment and Propagation Factors of Short-Wave Range in Estonia. 2016.

111. **Syed Saif Abrar**. Comprehensive Abstraction of VHDL RTL Cores to ESL SystemC. 2016.

112. **Arvo Kaldmäe**. Advanced Design of Nonlinear Discrete-time and Delayed Systems. 2016.

113. **Mairo Leier**. Scalable Open Platform for Reliable Medical Sensorics. 2016.

114. **Georgios Giannoukos**. Mathematical and Physical Modelling of Dynamic Electrical Impedance. 2016.

115. **Aivo Anier**. Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems. 2016.

116. **Denis Firsov**. Certification of Context-Free Grammar Algorithms. 2016.

117. **Sergei Astatpov**. Distributed Signal Processing for Situation Assessment in Cyber-Physical Systems. 2016.

118. **Erkki Moorits**. Embedded Software Solutions for Development of Marine Navigation Light Systems. 2016.

119. **Andres Ojamaa**. Software Technology for Cyber Security Simulations. 2016.

120. **Gert Toming**. Fluid Body Interaction of Biomimetic Underwater Robots. 2016.

121. **Kadri Umbleja**. Competence Based Learning – Framework, Implementation, Analysis and Management of Learning Process. 2017.

122. **Andres Hunt**. Application-Oriented Performance Characterization of the Ionic Polymer Transducers (IPTs). 2017.