Guarded recursion in the topos of trees
Internship report (updated version)

Bálint Kocsis

March 12, 2025

Abstract

Step-indexing is a semantic tool for stratifying circular, non-wellfounded definitions. The main idea is to use sequences of successive approximations to construct certain objects (types, propositions, functions, etc.), where the n-th element of a sequence corresponds to an approximation of the object if only the next n steps of computation are concerned. This can be used to construct models of type systems with complicated features such as general recursive types or general references.

The idea of step-indexing can be formalized in a logical, or type-theoretical, setting, by introducing a modality, called later, which allows us to talk about the next computation step.

In this work, we give an introduction to the logic of step-indexing, with a particular emphasis on its semantics in the topos of trees. We investigate sound rules in the internal logic of the topos of trees for commuting the later modality with quantifiers. Furthermore, we provide a Coq mechanization of the topos of trees, along with a shallow embedding of its internal logic.

Contents

1 Introduction

In the field of theoretical computer science, especially programming language semantics, it is often natural to use recursive, or self-referential, definitions. This observation applies both to recursively defined functions and predicates, and recursively defined types (domains). Sometimes, however, these definitions are not wellfounded when understood from a classical, set-theoretical perspective, in which case the existence or uniqueness of a solution is nontrivial, or even false.

This can happen for instance when the recursion variable occurs in a negative position. Suppose that A is a set with at least two elements, and we want to find a set X satisfying the equation

$$X = X \to A \tag{1}$$

where $X \to A$ is the set of functions from X to A. It follows from simple cardinal arithmetic that no such X exists. Hence, the equation cannot act as the *definition* of X.

1.1 Step-indexing

Step-indexing is a semantic tool for stratifying circular, non-wellfounded definitions. The main idea is to use sequences of successive approximations to construct certain objects (types, propositions, functions, etc.), where the n-th element of a sequence corresponds to an approximation of the object if only the next n steps of computation are concerned. The sequence is typically obtained by recursion on n, referred to as a *step-index*, thus making the definition wellfounded. Continuing our previous example, we may define a sequence X_n of sets by letting X_0 be any singleton set and setting $X_{n+1} = X_n \to A$.

The first application of step-indexing is due to Appel and McAllester [3], who presented a model for general recursive types for foundational proof-carrying code [2]. Later, Ahmed used step-indexing in her PhD thesis [1] to give semantics for programming languages with dynamically allocated higher-order store, such as general ML-like references, that may contain values of any type. Other projects that employ the technique include the Verified Software Toolchain [34], and the program logic Iris [22].

1.2 Guarded recursion and the later modality

Although a powerful technique, employing step-indexing directly is tedious, and results in more complicated constructions than standard set-theoretical models. Appel et al. [4] noticed that the details of step-indexing may be hidden using a modality $\blacktriangleright$, called 'later' and originally introduced by Nakano [27]. Intuitively, the $\blacktriangleright$ modality allows us to talk about data we only have access to in the next computation step. This modality is then used to guard self-references in recursive type definitions; hence the name *guarded recursion*. For instance, the guarded recursive version of Equation (1) is expressed as

$$X = \blacktriangleright X \to A$$

Furthermore, in his seminal paper [27], Nakano observed that fixed point combinators in lambda calculus may adequately be given the type $(\blacktriangleright X \to X) \to X$ without violating termination. Hence, $\blacktriangleright$ also allows the definition of guarded recursive functions.

Under the Curry-Howard correspondence, the $\blacktriangleright$ type former corresponds to a logical connective $\triangleright$, also pronounced 'later'. This leads us to consider step-indexed logics, where the truth of a proposition depends on a step-index which may be viewed as the number of computation steps left. Under this interpretation, $\triangleright P$ holds at $n+1$ if P holds at n. There is also an induction principle, called Löb induction, which mirrors the type of the guarded fixed point operators:

$$\triangleright P \supset P \vdash P$$

Intuitively, this principle states that in order to prove a proposition P, we may assume as an induction hypothesis that P holds one step later.

Birkedal et al. [10, 11] presented the topos of trees as a model of guarded recursion, and proposed to use its internal logic to construct synthetic versions of step-indexed models of programming languages and program logics in order to hide the step-indexing. As mentioned in Section 1.1, Iris [22]

is a program logic with a step-indexed model. In its current implementation, the mathematical framework for step-indexing is provided by ordered families of equivalences (OFEs) [22]. Hence, it is natural to ask if a synthetic model of Iris could be constructed in the internal logic of the topos of trees.

1.3 Rules for the $\triangleright$ modality

If we wish to use the internal logic of the topos of trees to construct a model of Iris, we need to ensure that the logic is rich enough to prove the soundness of the inference rules of Iris. For instance, consider the Iris rule

$$\triangleright(P * Q) \dashv\vdash \triangleright P * \triangleright Q$$

Since the definition of the separating conjunction $*$ contains an existential quantifier, it seems likely that we need rules to move the $\triangleright$ modality under an existential quantifier. However, the following rule is not sound:

$$\triangleright(\exists x : A.\, P) \dashv\vdash \exists x : A.\, \triangleright P$$

Finding as general rules as possible is also an interesting question on its own right. Unfortunately, to the best of our knowledge, no general rule commuting $\triangleright$ with quantifiers appears in the literature.

For this purpose, we propose a novel rule for step-indexed logics, from which previously known commuting rules are derivable. Our insights are based on an observation already made by Birkedal et al. ([11], Section 2.3) that the $\triangleright$ modality may be decomposed as $\triangleright = \texttt{lift} \circ \texttt{next}$. (The name `lift` was introduced in Clouston et al. [15], Section 3.1.)

1.4 Formalization

It seems that not much of the structure of the topos of trees has been formalized in the literature. To fill this hole, we formalize most constructions, definitions, and propositions in the present report. The original goal of the formalization was to experiment with a model of Iris constructed using the internal logic of the topos of trees, as alluded to in Section 1.3. This is however left to future work (see Section 6).

An important insight gained from the formalization concerns the representation of finite types. Our experiments have shown that the traditional inductive representation of the natural number indexed family `fin n` of finite types seemed to entail messy type coercions. Representing elements of `fin n` as natural numbers equipped with a proof that they are less than n proved to be more fruitful. The details are given in Section 4.1.

1.5 Contributions

Our contributions are the following:

- We offer an introduction to guarded recursion and the logic of step-indexing, with a particular emphasis on their semantics in the topos of trees. Our exposition is aimed at a wide audience, and attempts to provide intuition wherever possible. A reader with basic knowledge of logic and programming should be able to follow the gist of the text, although a bit of category theory could be useful.
- We present a novel axiom from which previously known inference rules exchanging the $\triangleright$ modality with a quantifier can be derived. Our axiom relies on the observation that we can decompose the $\triangleright$ modality into two components, and then study the interaction between the quantifiers and the two components separately.
- Finally, we provide a formalization of the topos of trees in the Coq proof assistant, including a shallow embedding of its internal logic. We provide details and technical insights in the report.

In Section 2, we review the main semantic definitions in the topos of trees. Then we study its internal logic in Section 3. In Section 4, we elaborate on some design choices and technical challenges of our Coq formalization. Related work is discussed in Section 5. Finally, Section 6 concludes with a summary and future directions.

2 The topos of trees

We begin by introducing the topos of trees as a model of guarded recursive functions. It helps if the reader is familiar with basic notions from category theory, although we aim to also explain the necessary bits in more elementary terms. In Section 2.1, we review the basic categorical properties of the topos of trees. Section 2.2 presents some important components that can be used to define guarded recursive functions.

2.1 Basic structure

Definition 2.1.1. The topos of trees $\mathcal{S}$ is the category of presheaves on ω, the set of natural numbers with their usual ordering.

Explicitly, this means that an object X in $\mathcal{S}$ is a family of sets $(X_n)_{n\in\omega}$ indexed by the natural numbers, together with restriction maps $(r^X_n \colon X_{n+1} \to X_n)_{n\in\omega}$. This can be depicted as the diagram

$$X_0 \xleftarrow{r^X_0} X_1 \xleftarrow{r^X_1} X_2 \xleftarrow{r^X_2} \cdots$$

A morphism f from X to Y is a family of maps $(f_n \colon X_n \to Y_n)_{n\in\omega}$ commuting with the restriction maps, as shown by the following commutative diagram:

$$\begin{array}{ccccccc} X_0 & \xleftarrow{r^X_0} & X_1 & \xleftarrow{r^X_1} & X_2 & \xleftarrow{r^X_2} & \cdots \\ \downarrow{\scriptstyle f_0} & & \downarrow{\scriptstyle f_1} & & \downarrow{\scriptstyle f_2} & & \\ Y_0 & \xleftarrow[r^Y_0]{} & Y_1 & \xleftarrow[r^Y_1]{} & Y_2 & \xleftarrow[r^Y_2]{} & \cdots \end{array}$$

For $n \leq m$, $r^X_{n,m} \colon X_m \to X_n$ is defined to be the composite $r_n \circ \cdots \circ r_{m-1}$. Moreover, if $x \in X_m$ and $n \leq m$, we use the notation $x|_n$ for $r^X_{n,m}(x)$, referred to as the *restriction of x to n*.

Intuitively, the topos of trees provides a setting in which sets can evolve over time. Given an object X, the set X_n can be thought of as the observable part of X after n steps. Here, "step" is an abstract notion, but in the case of programming it more or less corresponds to computation steps. To reinforce this intuition, we consider a non-trivial example of objects in the topos of trees.

Example 2.1.2. We define the object $\mathtt{Str}$ of *step-indexed streams* of natural numbers as follows:

$$\mathbb{N} \xleftarrow{r^{\mathtt{Str}}_0} \mathbb{N}^2 \xleftarrow{r^{\mathtt{Str}}_1} \mathbb{N}^3 \xleftarrow{r^{\mathtt{Str}}_2} \cdots$$

where $r^{\mathtt{Str}}_n \colon \mathbb{N}^{n+2} \to \mathbb{N}^{n+1}$ is the projection $(s_0, \ldots, s_n, s_{n+1}) \mapsto (s_0, \ldots, s_n)$. At step-index n we only have access the first $(n+1)$ elements of the stream, and thus we set $\mathtt{Str}_n = \mathbb{N}^{n+1}$. The restriction map $r^{\mathtt{Str}}_n$ expresses the connection between the views of the *same* stream at two consecutive time steps: if the view at index $(n+1)$ is $(s_0, \ldots, s_n, s_{n+1})$, then the view at index n is $(s_0, \ldots, s_n)$.

The category $\mathcal{S}$, being a topos, has many useful categorical and logical properties (for an introduction, see e.g. [23, 24]). The following definition collects the most important of these properties.

Definition 2.1.3.

(1) The initial object $\mathbf{0}$ and the terminal object $\mathbf{1}$ are the empty set and the singleton set at every level, respectively, with identities as restriction maps. We will denote both the unique morphism from $\mathbf{0}$ and the one to $\mathbf{1}$ by $!$.

(2) These are special cases of what are called *constant objects*, which form the image of the diagonal functor $\Delta \colon \mathbf{Set} \to \mathcal{S}$. For a set X, ΔX is the constant presheaf at X with $(\Delta X)_n = X$ and identities as restriction maps. We will usually omit the application of this functor, and write X for the object ΔX and f for the morphism Δf as well.

(3) The product of X and Y is the family $(X \times Y)_n = X_n \times Y_n$ together with restriction maps $r_n^X \times r_n^Y : X_{n+1} \times Y_{n+1} \to X_n \times Y_n$. Similarly, their coproduct has $(X + Y)_n = X_n + Y_n$ and $r_n^X + r_n^Y : X_{n+1} + Y_{n+1} \to X_n + Y_n$ as restriction maps. The projections and coprojections will be denoted by π_1, π_2 and κ_1, κ_2, respectively. We will use the notation (f, g) and $[f, g]$ for the unique morphism arising from the universal property of the product and coproduct, respectively.

(4) Given X and Y, the n-th component of the exponential Y^X is the set of $(n+1)$-tuples $(f_0, \dots, f_n)$, where $f_i : X_i \to Y_i$, commuting with the restriction maps. In other words, an element of $(Y^X)_n$ is like a morphism $X \to Y$ but only up to level n. The restriction maps are given by projections forgetting the last element. We will usually use the notation $X \Rightarrow Y$ for Y^X. We will write $\mathtt{ev}$ for the evaluation map $(X \Rightarrow Y) \times X \to Y$ and $\lambda f : Z \to (X \Rightarrow Y)$ for the exponential transpose of $f : Z \times X \to Y$. For a morphism $f : X \to Y$, $\ulcorner f \urcorner : \mathbf{1} \to (X \Rightarrow Y)$ is shorthand for $\lambda(f \circ \pi_2)$.

As an object of $\mathcal{S}$ consists of a family of sets (as opposed to a single set), we cannot talk about elements of an object directly. Instead, we have the following categorical definition:

Definition 2.1.4. A *global element* of an object X is a morphism $\mathbf{1} \to X$.

Equivalently, a global element $x : \mathbf{1} \to X$ is a collection $(x_n)_{n \in \omega}$ such that $x_n \in X_n$ and $r_n^X(x_{n+1}) = x_n$ for all $n \in \omega$. This also makes sense intuitively: x is a sequence of more and more refined approximations, where x_n is our view of the element at time step n.

We tend to think of global elements $\mathbf{1} \to X$ as being real elements. Indeed, they correspond to closed terms of type X in the internal logic; see Definition 3.3.3.

Example 2.1.5.

(1) The infinite stream of natural numbers $\mathtt{nats} : \mathbf{1} \to \mathtt{Str}$ can be defined by $\mathtt{nats}_n = (0, \dots, n)$. In general, any infinite sequence $s \in \mathbb{N}^\omega$ can be represented as the global element $\bar{s} : \mathbf{1} \to \mathtt{Str}$, $\bar{s}_n = (s_0, \dots, s_n)$.

(2) The stream function $\mathtt{inc} : \mathbb{N}^\omega \to \mathbb{N}^\omega$, $\mathtt{inc}(s)_n = s_n + 1$ adds one to every element in a stream. This function can also be defined on approximations: we have functions

$$\mathtt{inc}_n : \mathtt{Str}_n \to \mathtt{Str}_n,\ (s_0, \dots, s_n) \mapsto (s_0 + 1, \dots, s_n + 1)$$

for every $n \in \omega$. Since they satisfy $\mathtt{inc}_n \circ r_n^{\mathtt{Str}} = r_n^{\mathtt{Str}} \circ \mathtt{inc}_{n+1}$, this gives us a morphism $\mathtt{inc} : \mathtt{Str} \to \mathtt{Str}$ in $\mathcal{S}$.

(3) One may wonder if it is possible to generalize the previous example to all stream functions $\mathbb{N}^\omega \to \mathbb{N}^\omega$. The answer is no, but it is possible for *causal functions*, that is, functions such that the n-th element of the output depends only on the first n elements of the input. Such functions $f : \mathbb{N}^\omega \to \mathbb{N}^\omega$ can be written as $f(s)_n = g_n(s_0, \dots, s_n)$ for some $g_n : \mathbb{N}^{n+1} \to \mathbb{N}$, and can be represented by

$$\bar{f} : \mathtt{Str} \to \mathtt{Str},\ \bar{f}_n(s_0, \dots, s_n) = (g_0(s_0), \dots, g_n(s_0, \dots, s_n))$$

We shall see in Section 2.2 how to encode some non-causal functions in $\mathcal{S}$.

(4) The *head* function $\mathtt{hd} : \mathtt{Str} \to \mathbb{N}$ is given by the components $\mathtt{hd}_n(s_0, \dots, s_n) = s_0$.

In programming-motivated examples, we prefer to use the syntax of simply-typed lambda calculus, extended with appropriate type formers, to write terms. These are interpreted in the cartesian closed category $\mathcal{S}$ as usual [17, 31]. In particular, the interpretation of a term $\Gamma \vdash t : A$ of type A in context Γ is a morphism $[\![t]\!] : [\![\Gamma]\!] \to [\![A]\!]$, where $[\![\Gamma]\!]$ is the product of the interpretations of the types in Γ, and a closed term of type A is interpreted as a global element of $[\![A]\!]$. For instance, the term $f : \mathbb{N} \to \mathbb{N} \vdash \lambda x : \mathbb{N}.\, f\,(\mathtt{succ}\, x) : \mathbb{N} \to \mathbb{N}$ denotes the morphism $\lambda(\mathtt{ev} \circ (\mathtt{id} \times \mathtt{succ})) : (\mathbb{N} \Rightarrow \mathbb{N}) \to (\mathbb{N} \Rightarrow \mathbb{N})$, where $\mathtt{succ}$ is the successor function on natural numbers. The semantics of lambda terms (among other things) is described in more detail in Section 3.3 (Definition 3.3.3).

2.2 Guarded recursion

We have seen in the previous section (Example 2.1.5) that we can define causal stream functions as morphisms $\mathtt{Str} \to \mathtt{Str}$. Unfortunately, causal functions are not enough to work with streams. In particular, one of the defining coalgebraic operations of streams, the *tail function* $\mathtt{tl}\colon \mathbb{N}^\omega \to \mathbb{N}^\omega$ (also known as *derivative* [28]), given by $\mathtt{tl}(s)_n = s_{n+1}$, is not causal.

The reason why we cannot have a tail operation in the form of a morphism $\mathtt{Str} \to \mathtt{Str}$ is already apparent at level zero: we only have access to the first element of the stream, but the first element of the output should be the second element of the input. To resolve this mismatch, we need something which allows us to shift the step-indices.

This is achieved by introducing a new type former $\blacktriangleright$, which semantically corresponds to an endofunctor on $\mathcal{S}$. Then, intuitively, the type $\blacktriangleright X$ expresses that we will only have access to the underlying value of type X one step later. It turns out that this is also a key idea for implementing *guarded recursion*, that is, recursion on the available number of steps.

Definition 2.2.1. The functor $\blacktriangleright\colon \mathcal{S} \to \mathcal{S}$, called *later*, sends an object X of $\mathcal{S}$ to

$$\{*\} \xleftarrow{!} X_0 \xleftarrow{r_0^X} X_1 \xleftarrow{r_1^X} \cdots$$

That is, $(\blacktriangleright X)_0 = \{*\}$, $(\blacktriangleright X)_{n+1} = X_n$, $r_0^{\blacktriangleright X}$ is the unique map to a singleton, and $r_{n+1}^{\blacktriangleright X} = r_n^X$. The action of $\blacktriangleright$ on a morphism $f\colon X \to Y$ is given by $(\blacktriangleright f)_0 = \mathtt{id}_{\{*\}}$ and $(\blacktriangleright f)_{n+1} = f_n$.

Example 2.2.2.

(1) With the help of the $\blacktriangleright$ functor, we can now define the tail operation as a morphism $\mathtt{tl}\colon \mathtt{Str} \to \blacktriangleright\mathtt{Str}$, given by $\mathtt{tl}_0(s_0) = *$ and $\mathtt{tl}_{n+1}(s_0, s_1, \ldots, s_{n+1}) = (s_1, \ldots, s_{n+1})$.

(2) The *cons* operation $\mathtt{cons}\colon \mathbb{N} \times \mathbb{N}^\omega \to \mathbb{N}^\omega$ sends (a, s) to the infinite sequence $(a, s_0, s_1, \ldots)$. We could represent this operation as a morphism $\mathbb{N} \times \mathtt{Str} \to \mathtt{Str}$ in $\mathcal{S}$. However, a better type for $\mathtt{cons}$ would be $\mathbb{N} \times \blacktriangleright\mathtt{Str} \to \mathtt{Str}$. Intuitively, this type expresses that we only need the first $(n-1)$ elements of the input to compute the first n elements of the output. Thus, we define $\mathtt{cons}\colon \mathbb{N} \times \blacktriangleright\mathtt{Str} \to \mathtt{Str}$ as $\mathtt{cons}_0(a, *) = a$ and $\mathtt{cons}_{n+1}(a, (s_0, \ldots, s_n)) = (a, s_0, \ldots, s_n)$. We will use the standard $::$ notation in programs for this operation.

Definition 2.2.3. We define the natural transformation $\mathtt{next}\colon \mathtt{id}_{\mathcal{S}} \to \blacktriangleright$, whose components $\mathtt{next}_X\colon X \to \blacktriangleright X$ are given by $(\mathtt{next}_X)_n = r_n^{\blacktriangleright X}$. This is a well-defined family of morphisms in $\mathcal{S}$ since $\mathtt{next}_X$ respects the restrictions maps, as shown below:

$$\begin{array}{ccccccc}
X_0 & \xleftarrow{r_0^X} & X_1 & \xleftarrow{r_1^X} & X_2 & \xleftarrow{r_2^X} & \cdots \\
\downarrow{\scriptstyle !} & & \downarrow{\scriptstyle r_0^X} & & \downarrow{\scriptstyle r_1^X} & & \\
\{*\} & \xleftarrow[!]{} & X_0 & \xleftarrow[r_0^X]{} & X_1 & \xleftarrow[r_1^X]{} & \cdots
\end{array}$$

Essentially, $\mathtt{next}_X$ is the reflection of the restriction maps of X (or rather $\blacktriangleright X$) into $\mathcal{S}$. This makes it possible to talk about the internal structure of objects using the internal logic of $\mathcal{S}$. Proposition 3.4.4 will provide an illustration of this point.

Proposition 2.2.4. *There is a natural isomorphism $\blacktriangleright(X \times Y) \cong \blacktriangleright X \times \blacktriangleright Y$ induced by the morphism $(\blacktriangleright\pi_1, \blacktriangleright\pi_2) : \blacktriangleright(X \times Y) \to \blacktriangleright X \times \blacktriangleright Y$. In other words, the functor $\blacktriangleright$ preserves products.*

Proof. Let $l_{X,Y}\colon \blacktriangleright X \times \blacktriangleright Y \to \blacktriangleright(X \times Y)$ be given by $l_0(*, *) = *$, $l_{n+1}(x, y) = (x, y)$. This provides an inverse to $(\blacktriangleright\pi_1, \blacktriangleright\pi_2)$. □

Definition 2.2.5. Define $J_{X,Y}\colon \blacktriangleright(X \Rightarrow Y) \to (\blacktriangleright X \Rightarrow \blacktriangleright Y)$ to be the exponential transpose of the composite

$$\blacktriangleright(X \Rightarrow Y) \times \blacktriangleright X \xrightarrow{l_{X \to Y, X}} \blacktriangleright((X \Rightarrow Y) \times X) \xrightarrow{\blacktriangleright \mathtt{ev}} \blacktriangleright Y$$

Explicitly, the 0-th component of $J_{X,Y}$ sends the unique point of the domain to $\mathtt{id}_{\{*\}}$, and its $(n+1)$-st component maps $(f_0, \ldots, f_n) \in (X \to Y)_n$ to $(\mathtt{id}_{\{*\}}, f_0, \ldots, f_n)$.

The previous two definitions give the structure necessary to make $\blacktriangleright$ an applicative functor [25]. In programs, we will use the $\circledast$ operator introduced in [25] to apply a function to an argument beneath a $\blacktriangleright$ (see e.g. Example 2.2.9).

A crucial ingredient for being able to make guarded recursive definitions is the existence of unique fixed points for morphisms. Certainly, not every morphism $X \to X$ will have a fixed point (e.g. $\mathbf{0} \to \mathbf{0}$). However, we have the following proposition.

Proposition 2.2.6. *Let $f\colon \blacktriangleright X \to X$ be a morphism of $\mathcal{S}$. Then there exists a unique $x\colon \mathbf{1} \to X$ such that $f \circ \mathtt{next}_X \circ x = x$.*

Proof. The global element $x\colon \mathbf{1} \to X$ is defined by recursion: $x_0 = f_0(*)$ and $x_{n+1} = f_{n+1}(x_n)$. One can check using induction on n that x is well-defined, that is, $r_n^X(x_{n+1}) = x_n$, and that $f_n((\mathtt{next}_X)_n(x_n)) = x_n$ holds.

To see the uniqueness of the fixed point, note that if $x\colon \mathbf{1} \to X$ is such that $f \circ \mathtt{next}_X \circ x = x$, then $x_0 = f_0(r_0^{\blacktriangleright X}(x_0)) = f_0(*)$ and $x_{n+1} = f_{n+1}(r_{n+1}^{\blacktriangleright X}(x_{n+1})) = f_{n+1}(r_n^X(x_{n+1})) = f_{n+1}(x_n)$. Thus, the definition of x was forced. □

We will write μf for the unique fixed point of $f\colon \blacktriangleright X \to X$. Note that the existence of such fixed points does not compromise consistency, as there is in general no morphism $\blacktriangleright X \to X$. In particular, there is no morphism $\blacktriangleright \mathbf{0} \to \mathbf{0}$, so the above proposition does not give us an inhabitant of the empty type. In contrast, this morphism does exist in the OFE model of Iris [22]. As a result, fixed points in Iris are designed differently: we can take the fixed point of an arbitrary map $X \to X$, provided we can prove that it is *contractive* (see [22], Section 4.2).

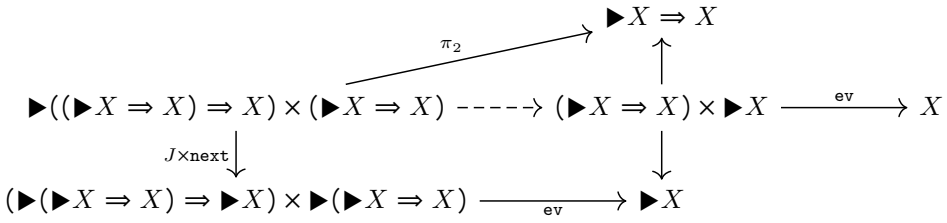
Example 2.2.7. Let $f\colon \blacktriangleright \mathtt{Str} \to \mathtt{Str}$ be the morphism given by the components $f_0(*) = 0$ and $f_{n+1}(s_0, \ldots, s_n) = (0, s_1, \ldots, s_n)$. Alternatively, f could also be defined as $f_n(s) = \mathtt{cons}_n(0, s)$ using `cons` from Example 2.2.2. Then $\mu f\colon \mathbf{1} \to \mathtt{Str}$ represents the constant stream with value 0.

Note that the construction of Proposition 2.2.6 is external, meaning that the assignment of the fixed point μf to the morphism f happens in the metalogic, instead of being an operation in $\mathcal{S}$. Thus, it does not correspond to a term of type $(\blacktriangleright X \Rightarrow X) \to X$ in lambda calculus. It turns out, however, that we can *define* such a term using this simple fixed point operator and other existing structure.

Definition 2.2.8. The morphism $\mathtt{fix}_X\colon (\blacktriangleright X \Rightarrow X) \to X$ is defined as follows:

$$\mathtt{fix}_X = \mathtt{ev} \circ (\mu(\lambda f) \circ !, \mathtt{id})$$

where $f\colon \blacktriangleright((\blacktriangleright X \Rightarrow X) \Rightarrow X) \times (\blacktriangleright X \Rightarrow X) \to X$ is the middle horizontal composite in the diagram



That is,

$$f = \mathtt{ev} \circ (\pi_2, \mathtt{ev} \circ (J \times \mathtt{next}))$$

If we write $\mu x : \blacktriangleright X.\, t$ for the fixed point of a term t of type X with one free variable x of type $\blacktriangleright X$, then the above definition can be expressed as the program

$$\mathtt{fix}(f) = (\mu g : \blacktriangleright((\blacktriangleright X \to X) \to X).\, \lambda f : \blacktriangleright X \to X.\, f\,(g \circledast \mathtt{next}\, f))\, f$$

To make notation easier, we adopt the fixed point operator of Definition 2.2.8 as a primitive in our informal programs, so that we can take fixed points of functions with parameters. Thus, if $\Gamma, x : \blacktriangleright X \vdash t : X$, then $\Gamma \vdash \mu x : \blacktriangleright X.\, t : X$ is interpreted as $\mathtt{fix}_X \circ \lambda t$ (see Definition 3.3.3).

Example 2.2.9.

(1) We show how to capture the morphism $\texttt{inc}\colon \texttt{Str} \to \texttt{Str}$ from Example 2.1.5 internally in lambda calculus. To this end, we first perform a series of transformations on the original definition.

The components $\texttt{inc}_n$ can also be defined by recursion on n as

$$\texttt{inc}_0(s_0) = s_0 + 1$$
$$\texttt{inc}_{n+1}(s_0, s_1, \ldots, s_{n+1}) = (s_0 + 1, \texttt{inc}_n(s_1, \ldots, s_{n+1}))$$

Employing the morphisms $\texttt{hd}$, $\texttt{tl}$, and $\texttt{cons}$ from Examples 2.1.5 and 2.2.2, this can be written as

$$\texttt{inc}_0(s) = \texttt{cons}_0(\texttt{hd}_0(s) + 1, \texttt{tl}_0(s))$$
$$\texttt{inc}_{n+1}(s) = \texttt{cons}_{n+1}(\texttt{hd}_{n+1}(s) + 1, \texttt{inc}_n(\texttt{tl}_{n+1}(s)))$$

If we agree to the convention that $\texttt{Str}_{-1} = \{*\}$ and $\texttt{inc}_{-1} = \texttt{id}_{\{*\}}$, then we can merge the previous two clauses into the equation

$$\texttt{inc}_n(s) = \texttt{cons}_n(\texttt{hd}_n(s) + 1, \texttt{inc}_{n-1}(\texttt{tl}_n(s)))$$

This recursive definition is then reflected in lambda calculus as the term

$$\texttt{inc} = \mu r : \blacktriangleright(\texttt{Str} \to \texttt{Str}).\, \lambda s : \texttt{Str}.\, (\texttt{hd}\, s + 1) :: (r \circledast \texttt{tl}\, s) : \texttt{Str} \to \texttt{Str}$$

(2) The constant stream with value zero (see Example 2.2.7) can be defined as

$$\texttt{zeros} = \mu s : \blacktriangleright\texttt{Str}.\, 0 :: s : \texttt{Str}$$

(3) We can use $\texttt{inc}$ to define the stream of natural numbers:

$$\texttt{nats} = \mu s : \blacktriangleright\texttt{Str}.\, 0 :: (\texttt{next}\, \texttt{inc} \circledast s) : \texttt{Str}$$

(4) A generalization of $\texttt{inc}$ is the program

$$\texttt{add} = \lambda n : \mathbb{N}.\, \mu r : \blacktriangleright(\texttt{Str} \to \texttt{Str}).\, \lambda s : \texttt{Str}.\, (\texttt{hd}\, s + n) :: (r \circledast \texttt{tl}\, s) : \mathbb{N} \to \texttt{Str} \to \texttt{Str}$$

that adds a specified natural number to every element of a stream. Observe that in this example, we take the fixed point of a function with a parameter, namely n.

We refer the reader to [15] for many more examples.

Remark 2.2.10. Although the $\blacktriangleright$ type former enables guarded recursion, it is not sufficient to express true coinductive types. For instance, the object $\texttt{Str}$ from Example 2.1.2 models guarded recursive streams, whereas coinductive streams are given by the constant object $\mathbb{N}^\omega$. Unfortunately, guarded recursive streams are not enough to represent all stream functions. In particular, using the system to be introduced in Section 3.3, we cannot define the function $\texttt{even}\colon \mathbb{N}^\omega \to \mathbb{N}^\omega$ keeping only elements at even indices.

3 Internal logic of the topos of trees

As every topos, $\mathcal{S}$ possesses a so called *internal logic* or *internal language* [21, 23] (also called Mitchell-Bénabou language [24]) that makes it possible to reason about $\mathcal{S}$ using higher-order logic. With the internal language, we can define objects and morphisms of $\mathcal{S}$ and reason about their properties just as if they were sets and functions. The informal lambda calculus notation we have been using so far is actually also part of the internal language, being the basis of the type theory that underlies the internal higher-order logic.

The reason this works is that we can define a notion of *internal truth* in a category (with suitable structure). In the case of toposes, the central component to achieve this is the *subobject classifier*, which, intuitively, is the object of truth values. Propositions and predicates are then identified with morphisms into the subobject classifier.

Before we tackle the internal logic of the topos of trees, we take look at a nonstandard interpretation of first-order logic, which should give a sense of the main ideas; this is the aim of Section 3.1. Section 3.2 then introduces the main logical definitions in $\mathcal{S}$. In Section 3.3, we formally present the syntax and semantics of the internal logic of $\mathcal{S}$. Finally, in Section 3.4, we discuss the interaction of the $\triangleright$ modality with the quantifiers.

3.1 Step-indexed propositions

Normally, propositions in classical logic are either true or false. That is, the semantic domain of propositions, i.e. the set of *truth values*, is the two element set $\{\texttt{true}, \texttt{false}\}$. By replacing this set and defining the meaning of logical connectives as operations on it, we arrive at other models of logic. For instance, we could try interpreting formulas as natural numbers. This would give the viewpoint that propositions are not simply true or false, but hold until a certain time or for a certain number of steps.

More precisely, let $\overline{\mathbb{N}} = \mathbb{N} \cup \{\omega\}$ be the set of natural numbers extended with an additional element ω which is greater than any natural number. We think of $n \in \mathbb{N}$ as the truth value 'something holds for n steps', the case $n = 0$ corresponding to the truth value *false*. The element ω is thought of as the truth value *true*, i.e. that 'something holds forever'. The interpretation of the propositional connectives is then the following:

$$\begin{aligned} \top &= \omega \\ \bot &= 0 \\ m \wedge n &= \min(m, n) \\ m \vee n &= \max(m, n) \\ m \supset n &= \begin{cases} n & \text{if } m > n \\ \omega & \text{otherwise} \end{cases} \end{aligned}$$

The conjunction of two propositions is true until both of them are true. Hence, if φ holds for m steps and ψ holds for n steps, then $\varphi \wedge \psi$ holds for $\min(m, n)$ steps. Disjunction is similar. The implication $\varphi \supset \psi$ holds until ψ holds whenever φ holds. If ψ holds for fewer steps than φ, the implication is only valid until ψ is valid; otherwise, the implication is valid forever.

Semantic entailment $m \vDash n$ is simply defined as the order relation on $\overline{\mathbb{N}}$. Thus, $\varphi \vDash \psi$ means ψ holds for at least as long as φ does. One can easily check that this interpretation is sound with respect to the standard inference rules of intuitionistic propositional logic. However, the law of excluded middle does not hold in this model: for $n > 0$ we have $n \vee \neg n = n$ (where $\neg n$ is $n \supset \bot$).

A slightly different perspective on this idea is to regard natural numbers not as the truth values themselves, but as time steps, and to define when a proposition holds at a certain time step. Thus, a proposition is identified with the set of time steps at which it holds. To be consistent with the previous interpretation, we impose the condition that if a proposition holds at a given step, it should also hold for all smaller time steps. This leads us to the following definition:

Definition 3.1.1. The set of truth values is the set

$$S = \{A \subseteq \mathbb{N} \mid \forall i, j \in \mathbb{N}.\, j \leq i \wedge i \in A \Rightarrow j \in A\}$$

of *downward closed subsets* of $\mathbb{N}$.

The interpretation of the propositional connectives is as follows:

$$\begin{aligned} \top &= \mathbb{N} \\ \bot &= \varnothing \\ A \wedge B &= A \cap B \\ A \vee B &= A \cup B \\ A \supset B &= \{i \in \mathbb{N} \mid \forall j \leq i.\, j \in A \Rightarrow j \in B\} \end{aligned}$$

It is straightforward to prove that if A and B are downward closed, so are $A \wedge B$ and $A \vee B$; $A \supset B$ is downward closed by definition. Semantic entailment is given by the subset relation.

In a classical metatheory, this model is equivalent to the previous one, in the sense that there is bijection $b \colon \overline{\mathbb{N}} \cong S$ preserving the interpretation of each of the connectives. Specifically, b sends $n \in \overline{\mathbb{N}}$ to the interval $[0..n) = \{m \in \mathbb{N} \mid m < n\}$; in particular, $b(0) = \varnothing$ and $b(\omega) = \mathbb{N}$. However, the latter interpretation is more constructive. This becomes clear once we consider quantifiers, which

are essentially infinitary versions of conjunction and disjunction. In the first model, they are given by

$$\forall P = \inf_{x \in X} P(x)$$
$$\exists P = \sup_{x \in X} P(x)$$

where X is a set (the domain of quantification) and $P \colon X \to \mathbb{N}$ is a predicate on X. Thus, the quantifiers are operations of type $(X \to \mathbb{N}) \to \mathbb{N}$. Unfortunately, if X is infinite, then the above definitions of these operations are uncomputable.

For formalization purposes in a constructive type theory, such as Coq or Agda, it is important that all functions be computable. This can be achieved in the second model, where the interpretation of the quantifiers is

$$\forall P = \bigcap_{x \in X} P(x) = \{i \in \mathbb{N} \mid \forall x \in X.\, i \in P(x)\}$$
$$\exists P = \bigcup_{x \in X} P(x) = \{i \in \mathbb{N} \mid \exists x \in X.\, i \in P(x)\}$$

for $P \colon X \to S$. These definitions are constructive since, type-theoretically, subsets of $\mathbb{N}$ are given by functions $\mathbb{N} \to \mathtt{Prop}$.

So far we have shown that there is model a of intuitionistic first-order logic in which formulas are interpreted as extended natural numbers (or, equivalently, downward closed subsets of $\mathbb{N}$). To end this section, we observe that the model supports an additional operation with meaningful logical content. Namely, it is the successor function $s \colon \overline{\mathbb{N}} \to \overline{\mathbb{N}}$ sending $n \in \mathbb{N}$ to $n+1$ and ω to itself. Under the equivalence $b \colon \overline{\mathbb{N}} \cong S$, this corresponds to the operation

$$\triangleright \colon S \to S,\ \triangleright(A) = \{i \in \mathbb{N} \mid i = 0 \vee (i > 0 \wedge i - 1 \in A)\} \tag{2}$$

referred to as the *later* modality. The logical explanation of $\triangleright$ is the following: $\triangleright(A)$ holds at step $n+1$ iff A holds at step n. Thus, $\triangleright$ allows us to shift the step-indices.

3.2 Logic in the topos of trees

We now revisit the development in the previous section, but inside the topos of trees. Firstly, we need an object Ω in $\mathcal{S}$ analogous to the set of truth values S (Definition 3.1.1). A naive choice might be the constant object ΔS. However, this does not work. Given a predicate $P \colon X \to \Delta S$, the compatibility requirement for morphisms implies $P_n = P_0 \circ r^X_{0,n}$ for all $n \in \mathbb{N}$. Thus, $P_0 \colon X_0 \to S$ determines all other components. Intuitively, this means that a predicate can only use information at level zero, which is too restrictive. The issue is that the step-indexed nature of objects in $\mathcal{S}$ is not taken into account.

Recall that an element $A \in S$ is thought of as the set of time steps at which a proposition holds, and that the n-th component of an object represents our view of that object with n available computation steps. This suggests that at level n we should consider time steps only up to n; that is, elements of Ω_n should be subsets of $[0..n]$. This turns out to be the right idea, justified by the fact that the resulting object is an instance of the categorical notion of a *subobject classifier*, which is the object of truth values in a category.

Definition 3.2.1. The *subobject classifier* Ω of $\mathcal{S}$ has components

$$\Omega_n = \{A \subseteq [0..n] \mid \forall i, j \in [0..n].\, j \leq i \wedge i \in A \Rightarrow j \in A\}$$

with restriction maps

$$r^\Omega_n(A) = A \cap [0..n]$$

Remark 3.2.2. Technically, the subobject classifier is not the object Ω itself, but the morphism $\mathbf{1} \xrightarrow{\mathtt{true}} \Omega$ that picks out the truth value 'true', defined below.

Example 3.2.3.

(1) We can lift a set-theoretic predicate $A \subseteq X$ to a morphism $\bar{A} \colon \Delta X \to \Omega$ given by

$$\bar{A}_n(x) = \begin{cases} [0..n] & \text{if } x \in A \\ \varnothing & \text{if } x \notin A \end{cases}$$

More abstractly, let $\mathbf{2} = \{\mathtt{true}, \mathtt{false}\}$ denote the set of booleans. There is a morphism $c \colon \mathbf{2} \to \Omega$ whose n-th component sends $\mathtt{true}$ to $[0..n]$ and $\mathtt{false}$ to $\varnothing$. Assuming a classical metatheory, subsets A of X correspond to *characteristic functions* $\chi_A \colon X \to \mathbf{2}$. Then $\bar{A}$ is simply described as the composite $X \xrightarrow{\chi_A} \mathbf{2} \xrightarrow{c} \Omega$.

Note that the above definition of $\bar{A}$ is nonconstructive: we make a case distinction on whether $x \in A$, which in general is undecidable. A constructive, albeit perhaps less intuitive, definition would be $\bar{A}_n(x) = \{i \in [0..n] \mid x \in A\}$.

(2) As a slightly more interesting example, consider the stream predicate

$$\mathtt{isZero} \colon \mathtt{Str} \to \Omega,\ \mathtt{isZero}_n(s) = \{i \in [0..n] \mid \forall j \leq i.\, s_j = 0\}$$

As the name suggests, `isZero` checks whether its argument stream is the constant zero stream. However, the truth value of the predicate is not a mere true or false, but the set of step-indices i such that the stream is constant zero up to i. Intuitively, if the first n elements of a stream are zero, then we can say that it is zero for n time steps.

We can carry over the set-theoretic definitions of the propositional connectives from Section 3.1, and obtain componentwise operations on Ω, and thus morphisms in $\mathcal{S}$.

Definition 3.2.4. The propositional connectives in $\mathcal{S}$ are defined as follows:

- True: $\mathtt{true} \colon \mathbf{1} \to \Omega$, $\mathtt{true}_n = [0..n]$
- False: $\mathtt{false} \colon \mathbf{1} \to \Omega$, $\mathtt{false}_n = \varnothing$
- Conjunction: $\mathtt{conj} \colon \Omega \times \Omega \to \Omega$, $\mathtt{conj}_n(A, B) = A \cap B$
- Disjunction: $\mathtt{disj} \colon \Omega \times \Omega \to \Omega$, $\mathtt{disj}_n(A, B) = A \cup B$
- Implication: $\mathtt{impl} \colon \Omega \times \Omega \to \Omega$, $\mathtt{impl}_n(A, B) = \{i \in [0..n] \mid \forall j \leq i.\, j \in A \Rightarrow j \in B\}$

The quantifiers are a bit trickier: since the domain of quantification is now a step-indexed family of sets, it might not be obvious which components to quantify over.

Definition 3.2.5. The universal quantifier in $\mathcal{S}$ is the morphism

$$\mathtt{all} \colon (X \Rightarrow \Omega) \to \Omega,\ \mathtt{all}_n(P) = \{i \in [0..n] \mid \forall j \leq i, x \in X_j.\, j \in P_j(x)\}$$

The existential quantifier in $\mathcal{S}$ is the morphism

$$\mathtt{ex} \colon (X \Rightarrow \Omega) \to \Omega,\ \mathtt{ex}_n(P) = \{i \in [0..n] \mid \exists x \in X_i.\, i \in P_i(x)\}$$

In the case of `all`, we need to quantify over the components at all lower step indices to make the resulting set downward closed. For `ex`, this is unnecessary: if $i \in P_i(x)$ for some $x \in X_i$, then $j \in P_j(x|_j)$ for all $j \leq i$ by the compatibility requirement for P. Note also that the type of predicates on an object X is given by the exponential $X \Rightarrow \Omega$ in $\mathcal{S}$, also called the *power object* of X, just like how predicates in the set-theoretical model of Section 3.1 are functions $X \to S$.

The later modality of Section 3.1 (Equation (2)) also has an analogue in $\mathcal{S}$ as a morphism $\Omega \to \Omega$. However, similarly to `cons` (Example 2.2.2), we can give it the stronger type $\blacktriangleright\Omega \to \Omega$, which is necessary for the definition of guarded recursive predicates. We use the name `lift` for the stronger version, and reserve $\triangleright$ for the original operation, following tradition.

Definition 3.2.6. We define the morphism $\mathtt{lift} \colon \blacktriangleright\Omega \to \Omega$, by

$$\begin{aligned} \mathtt{lift}_0(*) &= \{0\} \\ \mathtt{lift}_n(A) &= \{i \in [0..n] \mid i = 0 \vee (i > 0 \wedge i - 1 \in A)\} \quad (n > 0) \end{aligned}$$

Furthermore, $\triangleright \colon \Omega \to \Omega$ is defined as $\triangleright = \mathtt{lift} \circ \mathtt{next}$.

Finally, to prove properties of programs, we need an appropriate notion of equality in $\mathcal{S}$.

Definition 3.2.7. The *step-indexed equality* predicate $\mathtt{eq}\colon X \times X \to \Omega$ is given by

$$\mathtt{eq}_n(x, y) = \{i \in [0..n] \mid x|_i = y|_i\}$$

Example 3.2.8. The predicate `isZero` from Example 3.2.3 can be defined syntactically as

$$\mathtt{isZero} = \lambda s : \mathtt{Str}.\, s = \mathtt{zeros} : \mathtt{Str} \to \mathtt{Prop}$$

where `zeros` is the constant zero stream (see Example 2.2.9). Alternatively, we can also give a guarded recursive definition, equivalent to the previous one:

$$\mathtt{isZero} = \mu r : \blacktriangleright(\mathtt{Str} \to \mathtt{Prop}).\, \lambda s : \mathtt{Str}.\, \mathtt{hd}\, s = 0 \wedge \mathtt{lift}\,(r \circledast \mathtt{tl}\, s) : \mathtt{Str} \to \mathtt{Prop}$$

Note the use of `lift`. Intuitively, this is an application of the later modality: we only have access to the tail one step later, so any statements made about it must also refer to the future. However, we merely get a $\blacktriangleright\mathtt{Prop}$ from the recursive call instead of a `Prop`, so we must use `lift`.

3.3 Internal logic

In this section, we demonstrate how one can use the internal logic of $\mathcal{S}$ to reason about programs defined via guarded recursion. To start off, we formally introduce the language and inference rules of the logic, and describe its semantics in the topos of trees.

Definition 3.3.1. The internal logic $\mathcal{L}$ of $\mathcal{S}$ is a typed higher-order logic, whose types are generated by the grammar

$$A ::= X \mid \mathbf{1} \mid A \times A \mid \mathbf{0} \mid A + A \mid A \to A \mid \blacktriangleright A \mid \mathtt{Prop}$$

where X ranges over the objects of $\mathcal{S}$. Note that by including all objects of $\mathcal{S}$ as basic types, we get some redundancy. For instance, if $A, B \in \mathcal{S}$, then $A \times B$ arises both as a basic type and as the product of A and B. We identify all such pairs of types arising from the type constructors $\times$, $+$, $\to$ and $\blacktriangleright$. Furthermore, the types $\mathbf{1}$, $\mathbf{0}$, and `Prop` are identified with the objects $\mathbf{1}$, $\mathbf{0}$, and Ω, respectively.

The terms of $\mathcal{L}$ are defined by the grammar

$$\begin{aligned} t ::= {} & x \mid f \mid () \mid (t, t) \mid \pi_1 t \mid \pi_2 t \mid \mathtt{abort}\, t \mid \mathtt{inj}_1\, t \mid \mathtt{inj}_2\, t \mid \\ & \mathtt{case}\, t\, \mathtt{of}\, x.t; x.t \mid \lambda x : A.\, t \mid t\, t \mid \mathtt{next}\, t \mid t \circledast t \mid \mu x : A.\, t \mid \\ & \top \mid \bot \mid t =_A t \mid t \wedge t \mid t \vee t \mid t \supset t \mid \forall x : A.\, t \mid \exists x : A.\, t \mid \triangleright t \end{aligned}$$

where x ranges over a countable set of variables, and f ranges over the morphisms in $\mathcal{S}$. The `case` construct binds the variable x in the term following the dot, as do λ and μ-abstraction, and the quantifiers. Just like in the case of types, the inclusion of morphisms in $\mathcal{S}$ as term constants gives rise to duplicates in the syntax; these duplicates are identified.

A typing context is a finite list of distinct typed variables, written as $x_1 : A_1, \ldots, x_n : A_n$. The extension of Γ by a fresh variable x of type A is denoted by $\Gamma, x : A$. The typing judgment takes the form $\Gamma \vdash t : A$, where Γ is a typing context, t is a term, and A is a type. Its inference rules are the standard ones from the simply typed lambda calculus, extended with additional rules for the applicative structure of $\blacktriangleright$, the fixed point operator $\mu x : A.\, t$, and propositions in the expected way. They are displayed in the appendix (Figure 4).

The entailment relation of $\mathcal{L}$ is given by a judgment $\Gamma \mid P \vdash Q$, where Γ is a typing context and $\Gamma \vdash P, Q : \mathtt{Prop}$. This expresses that we can prove Q from the hypothesis P. We write $\Gamma \vdash P$ for $\Gamma \mid \top \vdash P$, and $\Gamma \mid P \dashv\vdash Q$ to mean $\Gamma \mid P \vdash Q$ and $\Gamma \mid Q \vdash P$. It includes standard structural rules, equality rules, and logical rules for the propositional connectives and quantifiers. The complete set of inference rules can be found in Figures 5 and 6 in the appendix.

Besides logical rules, we also have the familiar $\beta\eta$-laws for functions, products, and sums. These are complemented by equalities concerning the new type former $\blacktriangleright$ and its associated constructs, some of which are shown in Figure 1. The first two equations are the homomorphism and composition laws for applicative functors [25], while the third one establishes that $\mu x : \blacktriangleright A.\, t$ is indeed a fixed

$$\circledast\text{-NEXT}\quad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathtt{next}\, t \circledast \mathtt{next}\, u =_{\blacktriangleright B} \mathtt{next}\, (t\, u)}$$

$$\circledast\text{-COMP}\quad \frac{\Gamma \vdash f : \blacktriangleright(B \to C) \qquad \Gamma \vdash g : \blacktriangleright(A \to B) \qquad \Gamma \vdash t : \blacktriangleright A}{\Gamma \vdash f \circledast (g \circledast t) =_{\blacktriangleright C} \mathtt{next}\, \mathtt{comp} \circledast f \circledast g \circledast t}$$

$$\text{FIX-}\beta\quad \frac{\Gamma, x : \blacktriangleright A \vdash t : A}{\Gamma \vdash \mu x : \blacktriangleright A.\, t =_A t\,[x := \mathtt{next}\, (\mu x : \blacktriangleright A.\, t)]}$$

Figure 1: Term equalities for $\blacktriangleright$. $\mathtt{comp} : (B \to C) \to (A \to B) \to A \to C$ is function composition.

$$\triangleright\text{-INTRO}\quad \Gamma \mid P \vdash \triangleright P \qquad \triangleright\text{-MONO}\quad \frac{\Gamma \mid P \vdash Q}{\Gamma \mid \triangleright P \vdash \triangleright Q} \qquad \text{LÖB}\quad \frac{\Gamma \mid \triangleright P \vdash P}{\Gamma \vdash P} \qquad \wedge\text{-}\triangleright\quad \Gamma \mid \triangleright P \wedge \triangleright Q \vdash \triangleright(P \wedge Q)$$

$$\triangleright\text{-}\vee\quad \Gamma \mid \triangleright(P \vee Q) \vdash \triangleright P \vee \triangleright Q \qquad \supset\text{-}\triangleright\quad \Gamma \mid \triangleright P \supset \triangleright Q \vdash \triangleright(P \supset Q) \qquad \triangleright\text{-EQ}\quad \Gamma \mid \triangleright(t =_A u) \dashv\vdash \mathtt{next}\, t =_{\blacktriangleright A} \mathtt{next}\, u$$

Figure 2: Inference rules for $\triangleright$

point of t. There is also a corresponding η rule, expressing the uniqueness fixed points; see the appendix.

Figure 2 lists some important inference rules for the $\triangleright$ modality. Of particular interest is the rule LÖB, called *Löb induction*. This principle internalizes the technique of induction on the step-index, and it is the main tool for proving properties of guarded recursively defined programs. Note the formal similarity between LÖB and the fixed point operator of Proposition 2.2.6. Another important rule is $\triangleright$-EQ, which establishes a strong connection between the $\triangleright$ modality and the $\blacktriangleright$ type former.

It is worth mentioning that $\mathcal{L}$ also contains an inference rule corresponding to the principle of *propositional extensionality* (Figure 6), asserting that if two propositions entail each other, then they are equal. Furthermore, the principle of *function extensionality*, i.e. that two functions are equal whenever their values are equal for every argument, is a consequence of the η law for functions.

From the basic inference rules of $\mathcal{L}$, we can derive further sound reasoning principles. Some of these are presented in Figure 3. The rules commuting $\triangleright$ with the various connectives can easily

$$\text{STRONG-LÖB}\quad \Gamma \mid \triangleright P \supset P \vdash P \qquad \triangleright\text{-}\wedge\quad \Gamma \mid \triangleright(P \wedge Q) \vdash \triangleright P \wedge \triangleright Q \qquad \vee\text{-}\triangleright\quad \Gamma \mid \triangleright P \vee \triangleright Q \vdash \triangleright(P \vee Q)$$

$$\triangleright\text{-}\supset\quad \Gamma \mid \triangleright(P \supset Q) \vdash \triangleright P \supset \triangleright Q \qquad \exists\text{-}\triangleright\quad \Gamma \mid \exists x : A.\ \triangleright P \vdash \triangleright(\exists x : A.\, P)$$

$$\triangleright\text{-}\forall\quad \Gamma \mid \triangleright(\forall x : A.\, P) \vdash \forall x : A.\ \triangleright P$$

Figure 3: Derived rules for $\triangleright$

be proved using the monotonicity of $\triangleright$ and the introduction and elimination principles of the

connectives. The proof of STRONG-LÖB is less trivial, but it closely resembles the construction of the internal fixed point operator of Definition 2.2.8 from the external one.

For the semantics of the `case` construct, we need the following proposition which is a consequence of the cartesian closure of $\mathcal{S}$:

Proposition 3.3.2. *For all objects X, Y and Z of $\mathcal{S}$, the canonical map*

$$X \times Y + X \times Z \xrightarrow{[\mathrm{id}\times\kappa_1, \mathrm{id}\times\kappa_2]} X \times (Y + Z)$$

is an isomorphism.

Definition 3.3.3. We interpret types A as objects $[\![A]\!]$ of $\mathcal{S}$ in the standard way, using the structure of $\mathcal{S}$ described in the previous sections to interpret the type formers. The interpretation of typing contexts $\Gamma = x_1 : A_1, \ldots, x_n : A_n$ is defined as $[\![\Gamma]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$, associated to the left. In particular, the interpretation of the empty context is $\mathbf{1}$.

Terms $\Gamma \vdash t : A$ of type A in context Γ are interpreted as morphisms $[\![\Gamma]\!] \to [\![A]\!]$. In particular, closed terms of type A are interpreted as global elements of $[\![A]\!]$. The clauses of the interpretation are as follows (dropping the contexts and the types from the notation):

- If $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, $[\![x_i]\!]$ is the i-th projection $[\![\Gamma]\!] \to [\![A_i]\!]$ corresponding to the variable $(x_i : A_i) \in \Gamma$;
- For a morphism $f \colon A \to B$ in $\mathcal{S}$, $[\![f]\!]$ is $[\![\Gamma]\!] \to \mathbf{1} \xrightarrow{\ulcorner f \urcorner} ([\![A]\!] \Rightarrow [\![B]\!])$;
- $[\![()]\!] = !\colon [\![\Gamma]\!] \to \mathbf{1}$;
- $[\![\top]\!]$ is $[\![\Gamma]\!] \to \mathbf{1} \xrightarrow{\mathtt{true}} \Omega$; similarly for $[\![\bot]\!]$ using `false`;
- The unary operators π_1, π_2, `abort`, $\mathtt{inj}_1$, $\mathtt{inj}_2$, `next`, and $\triangleright$ are interpreted via postcomposition with $\pi_1, \pi_2, !, \kappa_1, \kappa_2$, `next`, and $\triangleright$, respectively;
- $[\![(t, u)]\!] = ([\![t]\!], [\![u]\!])$;
- $[\![t\, u]\!] = \mathtt{ev} \circ ([\![t]\!], [\![u]\!])$; the binary operators $=, \wedge, \vee$, and $\supset$ are interpreted similarly, replacing `ev` with `eq`, `conj`, `disj`, and `impl`, respectively;
- $[\![\mathtt{case}\, s\, \mathtt{of}\, x.t; y.u]\!]$ is the composite

$$[\![\Gamma]\!] \xrightarrow{(\mathrm{id}, [\![s]\!])} [\![\Gamma]\!] \times ([\![A]\!] + [\![B]\!]) \cong [\![\Gamma]\!] \times [\![A]\!] + [\![\Gamma]\!] \times [\![B]\!] \xrightarrow{[[\![t]\!], [\![u]\!]]} [\![C]\!]$$

  where the middle map is the isomorphism from Proposition 3.3.2;
- $[\![\lambda x : A.\, t]\!] = \lambda [\![t]\!]$;
- $[\![t \circledast u]\!]$ is the composite

$$[\![\Gamma]\!] \xrightarrow{([\![t]\!], [\![u]\!])} \blacktriangleright([\![A]\!] \Rightarrow [\![B]\!]) \times \blacktriangleright [\![A]\!] \xrightarrow{J \times \mathrm{id}} (\blacktriangleright [\![A]\!] \Rightarrow \blacktriangleright [\![B]\!]) \times \blacktriangleright [\![A]\!] \xrightarrow{\mathtt{ev}} \blacktriangleright [\![B]\!]$$

- $[\![\mu x : \blacktriangleright A.\, t]\!] = \mathtt{fix} \circ \lambda [\![t]\!]$; $[\![\forall x : A.\, P]\!]$ and $[\![\exists x : A.\, P]\!]$ are similar, using `all` and `ex`, respectively, in place of `fix`.

Lastly, semantic entailment $\Gamma \mid P \vDash Q$, where $\Gamma \vdash P, Q : \mathtt{Prop}$, is defined as

$$\Gamma \mid P \vDash Q \text{ iff } \forall n \in \mathbb{N}, \gamma \in [\![\Gamma]\!]_n\, n \in [\![P]\!]_n(\gamma) \Rightarrow n \in [\![Q]\!]_n(\gamma)$$

One can check that all the inference rules of Figures 5 and 6 are valid with respect to this interpretation. We say that a formula $\Gamma \vdash P : \mathtt{Prop}$ *holds* or that it is *valid* (in $\mathcal{S}$) if $\Gamma \mid \top \vDash P$.

We are finally in a position to be able to prove a nontrivial property of a guarded recursive program. We use natural language to carry out the proof, which can readily be translated to a derivation in $\mathcal{L}$. We indicate the less obvious steps by reference to particular inference rules.

Example 3.3.4. Consider the program `add` from Example 2.2.9. We wish to prove

$$\mathtt{add}\,(m + n) =_{\mathtt{Str}\to\mathtt{Str}} \mathtt{comp}\,(\mathtt{add}\, m)\,(\mathtt{add}\, n)$$

for all $m, n : \mathbb{N}$, where $\mathtt{comp} = \lambda f : B \to C.\, \lambda g : A \to B.\, \lambda x : A.\, f\,(g\, x)$ is function composition. Since `add` was defined as a fixed point, we will eventually need to handle a recursive call. Hence, we use Löb induction (LÖB) to get an induction hypothesis

$$\triangleright(\mathtt{add}\,(m + n) =_{\mathtt{Str}\to\mathtt{Str}} \mathtt{comp}\,(\mathtt{add}\, m)\,(\mathtt{add}\, n))$$

By function extensionality, it suffices to prove the equality once both sides are applied to an argument stream $s : \mathtt{Str}$. Unfolding the definition of $\mathtt{add}$, and using the appropriate β-rules, including one application of FIX-β, the left hand side becomes

$$(\mathtt{hd}\, s + (m + n)) :: (\mathtt{next}\, (\mathtt{add}\, (m + n)) \circledast \mathtt{tl}\, s)$$

Similarly, after unfolding the right hand side (using that $\mathtt{hd}\, (a :: s) = a$ and $\mathtt{tl}\, (a :: s) = s$), we get

$$((\mathtt{hd}\, s + n) + m) :: (\mathtt{next}\, (\mathtt{add}\, m) \circledast (\mathtt{next}\, (\mathtt{add}\, n) \circledast \mathtt{tl}\, s))$$

The heads of the two streams are equal (by associativity and commutativity of addition), so it suffices to prove the equality of the tails. By $\triangleright$-EQ, our induction hypothesis is equivalent to

$$\mathtt{next}\, (\mathtt{add}\, (m + n)) =_{\blacktriangleright (\mathtt{Str} \to \mathtt{Str})} \mathtt{next}\, (\mathtt{comp}\, (\mathtt{add}\, m)\, (\mathtt{add}\, n))$$

Now we conclude the proof using $\circledast$-NEXT and $\circledast$-COMP:

$$\begin{aligned} \mathtt{next}\, (\mathtt{add}\, (m + n)) \circledast \mathtt{tl}\, s &= \mathtt{next}\, (\mathtt{comp}\, (\mathtt{add}\, m)\, (\mathtt{add}\, n)) \circledast \mathtt{tl}\, s \\ &= \mathtt{next}\, \mathtt{comp} \circledast \mathtt{next}\, (\mathtt{add}\, m) \circledast \mathtt{next}\, (\mathtt{add}\, n) \circledast \mathtt{tl}\, s \\ &= \mathtt{next}\, (\mathtt{add}\, m) \circledast (\mathtt{next}\, (\mathtt{add}\, n) \circledast \mathtt{tl}\, s) \end{aligned}$$

3.4 The $\triangleright$ modality and quantifiers

We have seen that $\triangleright$ commutes with the propositional connectives $\wedge$, $\vee$, and $\supset$. We are interested in establishing similar rules for the quantifiers, allowing us to move $\triangleright$ under a quantifier. The first guess for such rules might be as follows:

$\triangleright$-$\exists$-COMM
$\Gamma \mid \triangleright(\exists x : A.\, P) \dashv\vdash \exists x : A.\, \triangleright P$

$\triangleright$-$\forall$-COMM
$\Gamma \mid \triangleright(\forall x : A.\, P) \dashv\vdash \forall x : A.\, \triangleright P$

As the rules $\exists$-$\triangleright$ and $\triangleright$-$\forall$ (Figure 3) show, one direction of both equivalences is derivable in the system $\mathcal{L}$. In both cases, however, the other direction is not semantically valid. To explain why, we first characterize logical validity in $\mathcal{S}$ in a more intuitive way, known in the literature as *Kripke-Joyal semantics* [23, 24]. This characterization is formulated in terms of a *forcing relation* $\Vdash$: for $\Gamma \vdash P : \mathtt{Prop}$, $n \in \mathbb{N}$, and $\gamma \in [\![\Gamma]\!]_n$, we define $n \Vdash P(\gamma)$ iff $n \in [\![P]\!]_n(\gamma)$. Intuitively, $n \Vdash P(\gamma)$ means that P holds at step-index n, where the parameters of P have been substituted with the semantic values γ. If Γ is the empty context, we simply write $n \Vdash P$.

Proposition 3.4.1 (Kripke-Joyal semantics)**.** *The forcing relation satisfies the following clauses:*

$$\begin{aligned} n &\Vdash \top && \textit{always} \\ n &\Vdash \bot && \textit{never} \\ n &\Vdash (t =_A u)(\gamma) && \textit{iff} \quad [\![t]\!]_n(\gamma) = [\![u]\!]_n(\gamma) \\ n &\Vdash (P \wedge Q)(\gamma) && \textit{iff} \quad n \Vdash P(\gamma) \wedge n \Vdash Q(\gamma) \\ n &\Vdash (P \vee Q)(\gamma) && \textit{iff} \quad n \Vdash P(\gamma) \vee n \Vdash Q(\gamma) \\ n &\Vdash (P \supset Q)(\gamma) && \textit{iff} \quad \forall m \le n.\, m \Vdash P(\gamma|_m) \Rightarrow m \Vdash Q(\gamma|_m) \\ n &\Vdash (\exists x : A.\, P)(\gamma) && \textit{iff} \quad \exists a \in [\![A]\!]_n.\, n \Vdash P(\gamma, a) \\ n &\Vdash (\forall x : A.\, P)(\gamma) && \textit{iff} \quad \forall m \le n, a \in [\![A]\!]_m.\, m \Vdash P(\gamma|_m, a) \\ n &\Vdash (\triangleright P)(\gamma) && \textit{iff} \quad n = 0 \vee n - 1 \Vdash P(\gamma|_{n-1}) \end{aligned}$$

It is clear from the definitions that $\Gamma \mid P \models Q$ iff for all $n \in \mathbb{N}$ and $\gamma \in [\![\Gamma]\!]_n$, $n \Vdash P(\gamma)$ implies $n \Vdash Q(\gamma)$. In particular, a formula P holds in $\mathcal{S}$ iff $n \Vdash P(\gamma)$ for all $n \in \mathbb{N}$ and $\gamma \in [\![\Gamma]\!]_n$. Thus, we can use Proposition 3.4.1 to prove or disprove semantic entailments by inductively calculating the meaning of logical formulas. Coming back to the rule $\triangleright$-$\exists$-COMM, and assuming that Γ is the empty context, we have

$$n + 1 \Vdash \triangleright(\exists x : A.\, P) \text{ iff } \exists a \in [\![A]\!]_n.\, n \Vdash P(a)$$

and

$$n+1 \Vdash \exists x : A.\ \triangleright P \text{ iff } \exists a \in [\![A]\!]_{n+1}.\ n \Vdash P(a|_n)$$

We see that, semantically, the two sides quantify over different levels of A. Setting $A = \blacktriangleright\mathbf{0}$ and $P = \top$, we have $[\![A]\!]_0 = \{*\}$ and $[\![A]\!]_1 = \varnothing$, so the left hand side is true at step-index 1, while the right hand side is not. This shows that the left-to-right implication is not generally true.

Similarly, we calculate the meaning of the two sides of $\triangleright$-$\forall$-COMM:

$$n+1 \Vdash \triangleright(\forall x : A.\ P) \text{ iff } \forall m \le n, a \in [\![A]\!]_m.\ m \Vdash P(a)$$
$$n+1 \Vdash \forall x : A.\ \triangleright P \text{ iff } \forall m \le n, a \in [\![A]\!]_{m+1}.\ m \Vdash P(a|_m)$$

Now taking $A = \blacktriangleright\mathbf{0}$ and $P = \bot$ provides a counterexample to the validity of the right-to-left implication at step-index 1.

In both cases, the asymmetry arises from the fact that the two sides quantify over elements on different levels, and that the right hand side only talks about P holding for the restriction of something one lever higher. Note that this is no longer an issue if we assume that the restriction maps are surjective. This motivates the following definition.

Definition 3.4.2. An object X of $\mathcal{S}$ is called *total* if all the restriction maps r_n^X are surjective.

For a total object X, considered as a type in $\mathcal{L}$, we have $n+1 \Vdash \triangleright(\exists x : X.\ P)$ if and only if $n+1 \Vdash \exists x : X.\ \triangleright P$, and similarly for $\forall$. Since $0 \Vdash \triangleright(\forall x : X.\ P)$ and $0 \Vdash \forall x : X.\ \triangleright P$ always hold, we get that the rule $\triangleright$-$\forall$-COMM is valid for total objects X. For $\triangleright$-$\exists$-COMM to hold at level 0, we need the additional assumption that X_0 is inhabited. It is easy to see that for total objects X, this is equivalent to all X_n being inhabited, which is in turn equivalent to the existence of a morphism $\mathbf{1} \to X$, i.e. being inhabited by a global element.

Definition 3.4.3. An object X of $\mathcal{S}$ is *total and inhabited* if it is total and inhabited by a global element.

Thus, the rule $\triangleright$-$\exists$-COMM holds for total and inhabited X. It turns out that we can characterize this property in the internal logic.

Proposition 3.4.4. *An object X of $\mathcal{S}$ is total and inhabited iff* $\texttt{next}\colon X \to \blacktriangleright X$ *is internally surjective, that is, iff the formula*

$$\texttt{TI}(X) := \forall y : \blacktriangleright X.\ \exists x : X.\ \texttt{next}\, x =_{\blacktriangleright X} y$$

holds in $\mathcal{S}$.

Proof. We have

$$n \Vdash \texttt{TI}(X) \text{ iff } \forall m \le n, \forall y \in (\blacktriangleright X)_m, \exists x \in X_m.\ \texttt{next}_m(x) = y$$

Hence, $\texttt{TI}(X)$ holds in $\mathcal{S}$ iff all components of $\texttt{next}\colon X \to \blacktriangleright X$ are surjective. Since $\texttt{next}_{n+1} = r_n^X$, and $\texttt{next}_0\colon X_0 \to \{*\}$ is surjective iff X_0 is inhabited, the claim follows. □

We have thus shown the soundness of the following rules:

$$\frac{\vdash \texttt{TI}(A)}{\Gamma \mid \triangleright(\exists x : A.\ P) \dashv\vdash \exists x : A.\ \triangleright P}\ \triangleright\text{-}\exists\text{-COMM-TI} \qquad \frac{\vdash \texttt{TI}(A)}{\Gamma \mid \triangleright(\forall x : A.\ P) \dashv\vdash \forall x : A.\ \triangleright P}\ \triangleright\text{-}\forall\text{-COMM-TI}$$

It would be nice if we could derive these rules in the system $\mathcal{L}$. Unfortunately, this does not seem to be possible. Hence, there arises a natural question: is there a more general rule, working uniformly for all types A, from which $\triangleright$-$\exists$-COMM-TI and $\triangleright$-$\forall$-COMM-TI could be derived? We make two observations that will lead us to answer this question.

Firstly, as noted earlier (Definition 3.2.6), we can decompose the later modality as the composite $\triangleright = \texttt{lift} \circ \texttt{next}$. This suggests that we could study the interaction of the quantifiers with `lift` and `next` separately. In fact, the homomorphism property of `next` implies that `next` commutes with

the quantifiers. More precisely, let $\mathtt{ex}, \mathtt{all} : (X \Rightarrow \Omega) \to \Omega$ be the morphisms from Definition 3.2.5, which can be defined internally as

$$\mathtt{ex} = \lambda P : X \Rightarrow \Omega.\, \exists x : X.\, P\, x, \quad \text{and} \quad \mathtt{all} = \lambda P : X \Rightarrow \Omega.\, \forall x : X.\, P\, x$$

Then, by $\circledast$-NEXT, we have

$$\mathtt{next}\,(\exists x : A.\, P\, x) = \mathtt{next}\,(\mathtt{ex}\, P) = \mathtt{next}\,\mathtt{ex} \circledast \mathtt{next}\, P \tag{3}$$

and similarly

$$\mathtt{next}\,(\forall x : A.\, P\, x) = \mathtt{next}\,(\mathtt{all}\, P) = \mathtt{next}\,\mathtt{all} \circledast \mathtt{next}\, P$$

for every predicate $P : X \Rightarrow \Omega$. Hence, it seems that the difficulties are concentrated within the $\mathtt{lift}$ operation.

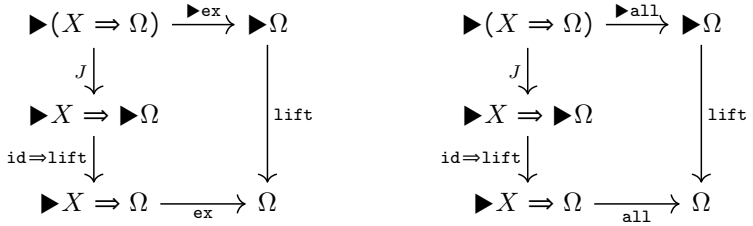
Secondly, recall that, semantically, the right hand side of the original rules quantifies over elements that are one level higher compared to the elements quantified over by the left hand side, due to the shift introduced by $\triangleright$. To resolve this mismatch, we could try also introducing a shift on the type level, by using $\blacktriangleright X$ instead of X as the domain of quantification on the right hand side.

From these considerations, we can guess reasonable candidates for sound rules that commute $\mathtt{lift}$ with a quantifier. Since the type of $\mathtt{lift}$ is $\blacktriangleright\Omega \to \Omega$, we start with a 'delayed' predicate $Q : \blacktriangleright(A \to \mathtt{Prop})$, and apply the quantifier under the $\blacktriangleright$. Thus, we arrive at the following rules:

$$\text{LIFT-}\exists\text{-COMM}\quad \frac{\Gamma \vdash Q : \blacktriangleright(A \to \mathtt{Prop})}{\Gamma \mid \mathtt{lift}\,(\mathtt{next}\,\mathtt{ex} \circledast Q) \dashv\vdash \exists y : \blacktriangleright A.\, \mathtt{lift}\,(Q \circledast y)}$$

$$\text{LIFT-}\forall\text{-COMM}\quad \frac{\Gamma \vdash Q : \blacktriangleright(A \to \mathtt{Prop})}{\Gamma \mid \mathtt{lift}\,(\mathtt{next}\,\mathtt{all} \circledast Q) \dashv\vdash \forall y : \blacktriangleright A.\, \mathtt{lift}\,(Q \circledast y)}$$

The soundness of these rules can be established using the original semantics (Definition 3.3.3). The proof essentially amounts to checking the commutativity of the diagrams below; we omit the details.



The rules $\triangleright$-$\exists$-COMM-TI and $\triangleright$-$\forall$-COMM-TI can indeed be derived from the new rules. We show how to do this for the existential quantifier; the universal quantifier is analogous. Given P and writing $\tilde{P} = \lambda x : A.\, P$, we have

$$\triangleright(\exists x : A.\, P) = \mathtt{lift}\,(\mathtt{next}\,(\exists x : A.\, \tilde{P}\, x)) = \mathtt{lift}\,(\mathtt{next}\,\mathtt{ex} \circledast \mathtt{next}\, \tilde{P})$$

by Equation (3). By LIFT-$\exists$-COMM, this holds iff $\exists y : \blacktriangleright A.\, \mathtt{lift}\,(\mathtt{next}\, \tilde{P} \circledast y)$. Since A is total and inhabited, this is equivalent to $\exists x : A.\, \mathtt{lift}\,(\mathtt{next}\, \tilde{P} \circledast \mathtt{next}\, x)$. But

$$\mathtt{lift}\,(\mathtt{next}\, \tilde{P} \circledast \mathtt{next}\, x) = \mathtt{lift}\,(\mathtt{next}\,(\tilde{P}\, x)) = \triangleright P$$

using $\circledast$-NEXT, so we are done. Note that the assumption $\mathtt{TI}(A)$ is only necessary for the left-to-right direction.

The generality of the new rules (LIFT-$\exists$-COMM and LIFT-$\forall$-COMM) indicates that $\mathtt{lift}$ should be taken as the primitive logical connective, with $\triangleright P = \mathtt{lift}\,(\mathtt{next}\, P)$ being a defined connective. This claim is also supported by the necessity of $\mathtt{lift}$ for defining guarded recursive predicates (Example 3.2.8). However, it is not yet clear what the analogues of e.g. monotonicity or Löb induction would be, or if such analogues exist. This question is further discussed in Section 6.

4 Coq formalization

We have formalized the topos of trees, along with the associated structure introduced so far, in the Coq proof assistant [32]. More specifically, the formalization encompasses the basic categorical structure (Section 2.1), the definitions necessary for guarded recursion (Section 2.2), and the logical components of Section 3.2. Moreover, we have verified semantically some of the inference rules of the internal logic, in particular the novel rules of Section 3.4.

The formalization does not contain the examples in this report. It is important to note that we do not define a syntax for the internal logic, as we did in Section 3.3. Instead, we work with the semantics directly, and use combinators to construct terms. The inference rules of the logic are then stated and proved as lemmas. In other words, we use a *shallow embedding* [35] of the internal logic. This contrasts with a *deep embedding* [35] of the syntax as an inductive definition, together with an interpretation function to show soundness, as was done in Definition 3.3.3.

In the following, we discuss some interesting aspects of the formalization, including some of the design choices and technical challenges. Section 4.1 motivates our definition of finite types. Section 4.2 shows how we work with proof-irrelevant propositions. Sections 4.3 and 4.4 explain some general techniques we used in our formalization via the examples of objects, morphisms, and exponentials. Section 4.5 expands on the concept of a shallow embedding and how we applied it to formalize the internal logic. Finally, Section 4.6 touches on the axioms we relied on and how their use could be eliminated.

4.1 Finite types

The definition of the subobject classifier (Definition 3.2.1) refers to the set $[0..n]$ of natural numbers from 0 to n. Therefore, we need to represent such finite sets. For this purpose, the Coq standard library defines an inductive family of types as follows:

```
Inductive fin : nat → Type :=
  | FZ {n} : fin (S n)
  | FS {n} : fin n → fin (S n).
```

Then, `fin n` corresponds to the finite set $[0..n) = \{0, \dots, n-1\}$: the indices in the types of the constructors ensure that `fin n` has exactly n canonical inhabitants.

Given `k : fin n`, we often need to treat `k` as a natural number, or as being an element of a bigger finite type, e.g. `fin (S n)`. In set theory, we can do this since $[0..n) \subseteq [0..n+1) \subseteq \mathbb{N}$; in Coq, however, every term has at most one type. Thus, we need conversion functions

```
fin_to_nat : ∀ {n}, fin n → nat
FW : ∀ {n}, fin n → fin (S n)
```

defined using the eliminator of the type family `fin`. But then it is not immediately clear that `FW k` and `k` represent the same natural number. In particular, equations such as

```
fin_to_nat (FW k) = fin_to_nat k
```
(4)

do not hold by definition and need to be proven by induction on `k`. Even worse, such lemmas are necessary to type certain simple expressions, an example of which is given in Section 4.4. This is extremely inconvenient and can become quite messy.

Our solution is to define `fin n` as the sigma type `{m : nat | m < n}`. This closely resembles the set-theoretic definition: `fin n` is the *subtype* of `nat` consisting of the natural numbers less than `n`. The main advantage of this representation is that `FW` only changes the second component, and `fin_to_nat` is simply the first projection. Thus, Equation (4) holds by definition.

To make working with finite types more convenient, we employ Coq's *definitionally proof-irrelevant* sort `SProp` of propositions [19]. Definitional proof-irrelevance means that if `A : SProp`, then any two inhabitants of `A` are definitionally equal. In the definition of `fin`, we replace the less-than relation `<` by its proof-irrelevant counterpart (see Section 4.2). The effect of this change is that two terms of type `fin n` are now convertible iff their first projections are convertible. Thus, once we define helper functions `FZ` for zero and `FS` for successor, we can treat elements of finite types as if they were natural numbers, with some additional type information.

We can further enhance the situation by declaring the conversion function `fin_to_nat` as a Coq coercion:

```
Coercion fin_to_nat : fin >-> nat.
```

This lets us leave applications of `fin_to_nat` implicit.

4.2 Proof-irrelevant order relations

To avoid defining custom order relations, we implement the proof-irrelevant versions of $\leq$ and $<$, denoted by $\preceq$ and $\prec$ respectively, using *propositional truncation* [33, Section 3.7], called `Squash` in Coq. That is, `x ⪯ y` stands for `Squash (x ≤ y)`, and `x ≺ y` means `S x ⪯ y` (which is the same as `Squash (x < y)`). Thus, we can easily prove lemmas such as

```
Sle_n : ∀ n, n ⪯ n
Sle_S : ∀ {n m}, n ⪯ m → n ⪯ S m
Sle_S_n : ∀ {n m}, S n ⪯ S m → n ⪯ m
```

by eliminating and introducing `Squash` and using the corresponding lemmas for $\leq$. Unfortunately, this method still requires some boilerplate, as we need to create a new lemma for every lemma we wish to use. It might be possible to derive such lemmas automatically by some form of tactic metaprogramming; we have not tried to do so.

In the next section, we will make use of the following large elimination principle for $\preceq$:

```
Sle_rect : ∀ n m (P : ∀ m, n ⪯ m → Type),
  P n (Sle_n n) → (∀ m (H : n ⪯ m), P m H → P (S m) (Sle_S H)) →
  ∀ H : n ⪯ m, P m H
```

Due to restrictions on how types in `SProp` can be eliminated, we have to define `Sle_rect` manually by recursion on `m` and `n`.

Remark 4.2.1. The necessity of defining large elimination for the order relation is not the consequence of using `SProp` instead of `Prop`. Since `Prop` cannot be eliminated into `Type` either, we would need to act similarly if we used $\leq$.

4.3 Objects and morphisms

In the previous sections, many definitions about the topos of trees were presented as one or more components satisfying certain properties. For instance, objects are families of types together with restriction functions; morphisms are families of functions satisfying a naturality condition.

In Coq, we can conveniently package up such definitions as record declarations. The type of objects is defined as follows:

```
Record Object : Type :=
  { obj :> nat → Type
  ; restr : ∀ n, obj (S n) → obj n
  }.
```

This creates a new record type along with the two corresponding record projections

```
obj : Object → nat → Type
restr : ∀ (X : Object) n, obj X (S n) → obj X n
```

The notation `obj :> nat → Type` automatically declares the record projection `obj` as a coercion. This allows us to write `X n` instead of `obj X n` for the `n`-th component of an object `X`. For readability, we keep the first argument of `restr` implicit.

We define the more general restriction operations $r^X_{n,m}$ (see Definition 2.1.1) by recursion on the proof of $n \preceq m$:

```
Definition restr' {X : Object} {m n} : n ⪯ m → X m → X n :=
  Sle_rect n m (λ m _, X m → X n) id (λ m _ f, f ∘ restr m).
```

As a special case of `restr'`, we also have

```
Definition restrTo {X : Object} {n} (i : [0..n]) (x : X n) : X i :=
  restr' (Sle_S_n (Spr2 i)) x.
```

where `[0..n]` is notation for `fin (S n)`, and `Spr2 i` is proof that `i ≺ S n` (recall that `fin n` is defined as a sigma type). This corresponds to the notation $x|_i$ introduced in Definition 2.1.1. This use case explains why we need to use `≼` instead of the ordinary ≤ for `restr'`.

Similarly to objects, morphisms are also defined as a record:

```
Record Morphism (X Y : Object) : Type :=
  { morph :> ∀ n, X n → Y n
  ; morph_natural : ∀ n (x : X (S n)),
      morph n (restr n x) = restr n (morph (S n) x)
  }.

Infix "⟶" := Morphism.
```

Notice how the formal definitions of both objects and morphisms follow closely the definitions given in the current report.

An important difference between objects and morphisms is that morphisms also contain a proof component. That is, whenever we want to construct a morphism `X ⟶ Y` (which is a family of maps `X n → Y n` for all `n`), we have the obligation to show that it satisfies the naturality condition `morph_natural`. Typically, we wish to carry out such verifications using Coq's tactic language instead of constructing proof terms by hand.

A convenient way to achieve this is provided by Coq's `Program` facilities, originally proposed by Sozeau [30]. The `Program Definition` command allows us to leave holes in a definition, which can then be filled in using Coq's proof mode. As an example, consider

```
Program Definition mcomp {X Y Z} (f : Y ⟶ Z) (g : X ⟶ Y) : X ⟶ Z :=
  ⟦λ n, f n ∘ g n⟧.
Next Obligation. ... Qed.
```

Here, `⟦h⟧` is notation for `{| morph := h; morph_natural := _ |}`, which is Coq syntax for constructing a record by listing its fields. The underscore is used to defer the proof of naturality, which is then provided using the `Next Obligation` command. If there are multiple holes, they can be solved one-by-one with consecutive applications of `Next Obligation`. We make extensive use of this pattern throughout the formalization.

4.4 Exponentials

Given `X Y : Object`, we may construct the exponential `Exp X Y`, denoted by `X ⇒ Y`, based on Definition 2.1.3. There, we have seen that the n-th component of the exponential $X \Rightarrow Y$ is the collection of $(n+1)$-tuples $(f_i \colon X_i \to Y_i)_{i \in [0..n]}$ commuting with the restriction maps. Therefore, similarly to `Morphism`, we define the `n`-th component of `Exp X Y` as a record:

```
Record Exp_obj (X Y : Object) (n : nat) : Type :=
  { Exp_morph :> ∀ i : [0..n], X i → Y i
  ; Exp_morph_natural : ∀ (i : [0..n|]) (x : X (S i)),
      Exp_morph (FW i) (restr i x) = restr i (Exp_morph (FS i) x)
  }.
```

Here, `[0..n|]` and `[0..n]` are notations for `fin n` and `fin (S n)`, respectively.

It is worth noting a subtle but important point regarding the definition of `Exp_obj`. In the expression `Exp_morph (FW i) (restr i x)` in the type of `Exp_morph_natural`, the second argument of `Exp_morph` is supposed to have type `X (fin_to_nat (FW i))`. However, `restr i x` has type `X (fin_to_nat i)`. Thus, the well-typedness of this expression crucially depends on Equation (4) in Section 4.1 holding definitionally. Had we chosen the alternative representation of `fin` as an inductive family, it would be necessary to transport along this equation explicitly. This would make reasoning with `Exp_morph_natural` (and in general, `Exp_morph`) much more difficult.

Similarly, the type of `x` in the expression `Exp_morph (FS i) x` is `X (S (fin_to_nat i))`, but its expected type is `X (fin_to_nat (FS i))`. Thus, the expression is well-typed only because the equation

```
fin_to_nat (FS i) = S (fin_to_nat i)
```
(5)

holds definitionally. However, this would also be the case with the inductive definition of `fin`, where Equation (5) would be one of the defining equations for `fin_to_nat`.

4.5 Logic

As stated in Definition 3.2.1, the n-th component of the subobject classifier is the collection of downward closed subsets of $[0..n]$. In type theory, subsets A of a set X are usually formalized as predicates `A : X → Prop`, where $x \in A$ iff `A x` holds. Hence, a reasonable formalization of Ω_n is

```
Record SOC_obj (n : nat) :=
  { SOC_pred :> [0..n] → Prop
  ; SOC_pred_closed' : ∀ (j i : [0..n]), j ≤ i → SOC_pred i → SOC_pred j
  }.
```

However, we can exploit the order structure on the natural numbers to simplify the closedness requirement. We replace the field `SOC_pred_closed'` by

```
SOC_pred_closed : ∀ i : [0..n|], SOC_pred (FS i) → SOC_pred (FW i)
```

That is, we only require $n+1 \in A \Rightarrow n \in A$ for $A \in \Omega_n$. The stronger condition `SOC_pred_closed'` then follows by induction on the proof of `j ≤ i` (using `Sle_rect`).

As mentioned in the introduction to this section, we use a shallow embedding to formalize the internal logic of $\mathcal{S}$ in Coq. This means that we use the semantics of the logic in $\mathcal{S}$ to represent types, terms, and proof derivations. That is, types are objects, (well-typed) terms are morphisms from their typing context to their type, and proof derivations are metatheoretic (Coq) proofs. The definitions of term constructors are then given by their interpretations according to Definition 3.3.3, whereas the inference rules are verified with respect to semantic entailment. In this way, we obtain a small combinator library with which we can construct and reason about terms as if they were syntactic objects.

For instance, we have the internal conjunction

```
conjI : Ω × Ω ⟶ Ω
```

introduced in Definition 3.2.4, where `Ω : Object` is the subobject classifier. We can define the respective term former as

```
Definition conj {Γ} (P Q : Γ ⟶ Ω) : Γ ⟶ Ω := conjI ∘ ⟨P, Q⟩.
```

where `⟨P, Q⟩` denotes categorical pairing. Note that `conj` corresponds to the typing rule CONJ (Figure 4): given two *semantic* terms `P` and `Q` of type `Ω` (i.e. propositions) in context `Γ`, it constructs another proposition in the same context. Also note that its definition follows the semantics given in Definition 3.3.3.

As another example, consider the universal quantifier, corresponding to the typing rule FORALL:

```
allI : ∀ {X}, (X ⇒ Ω) ⟶ Ω
Definition all {Γ} A (P : Γ × A ⟶ Ω) : Γ ⟶ Ω := allI ∘ λ(P).
```

where `λ(P)` is the exponential transpose of `P`. Here, `P` is a semantic proposition in context `Γ × A`, that is, `Γ` extended with a new variable of type `A`.

Given the embedding of our term language into Coq, all we have left to do is to verify semantically the inference rules of the logic. First, recall the semantic provability relation $\Gamma \mid P \models Q$ from Definition 3.3.3:

```
Definition entails {Γ} (P Q : Γ ⟶ Ω) : Prop :=
  ∀ n γ, P n γ n → Q n γ n.
```

The inference rules of the logic (Figures 5 and 6) can then be proved as separate lemmas. For example, the following lemma states conjunction introduction (∧-INTRO):

```
Lemma conj_intro {Γ} {R P Q : Γ ⟶ Ω} :
  R ⊢ P →
  R ⊢ Q →
  R ⊢ conj P Q.
```

Perhaps a bit confusingly, we use the ⊢ symbol for provability in the formalization. This is because we would like to treat the shallow embedding of the internal language as a combinator calculus, so we prefer to use syntactic notations.

An interesting aspect of our shallow embedding is that there are no variable names: we work with purely categorical combinators. Since terms are represented as morphisms from their context to their type, substitution can simply be implemented by precomposition. For instance, the rule ∀-INTRO is formalized as follows:

```
Lemma all_intro {Γ A} (R : Γ ⟶ Ω) (P : Γ × A ⟶ Ω) :
  R ∘ π₁ ⊢ P →
  R ⊢ all A P.
```

Here, the first projection π_1 on the left hand side of the premise is a weakening substitution. It is necessary to bring `R` into the same context as `P`, which has an additional free variable of type `A`. Another example is the rule ∀-ELIM:

```
Lemma all_elim {Γ A} (P : Γ × A ⟶ Ω) (t : Γ ⟶ A) :
  all A P ⊢ P ∘ ⟨mid, t⟩.
```

where `mid : ∀ {X}, X ⟶ X` is the identity morphism. In the conclusion, `⟨mid, t⟩` is the categorical version of the substitution $[x := t]$, mapping the last variable to `t` and leaving all other variables unchanged.

4.6 Axioms

In our formalization, we often have to show equality of morphisms, e.g. when proving categorical laws. Since morphisms are essentially families of functions, this involves showing equality of Coq functions. In most cases, such equalities do not hold by definition. Hence, we make use of the axiom of *function extensionality*, stating that two functions are equal iff they are equal at every input. This allows us for example to perform case distinction or induction on the input to show equality of the outputs.

In Section 4.3, we have seen that a morphism is actually a record, containing a family of functions *together with* a proof of their naturality. This means that proving equality of morphisms requires proving equality of both components, thus in particular, of equality proofs. Such higher equalities are notoriously difficult to deal with in constructive type theory. To simplify matters, we assume the *proof-irrelevance* axiom, which states that any two proofs of the same proposition are equal. This way, we can prove the following lemma, expressing that equality of morphisms follows from equality of their underlying families of functions:

```
Lemma morph_inj {X Y} {f g : X ⟶ Y} (e : morph f = morph g) : f = g.
```

The above two considerations also apply to the other record types, namely `Exp_obj` and `SOC_obj`. The first component of the latter is a function with codomain `Prop`. Thus, to prove equality of inhabitants of `SOC_obj` (necessary for e.g. naturality proofs of logical connectives), we need to prove equality of Coq propositions. For this, we use the *propositional extensionality* axiom, stating that two propositions are equal iff they imply each other.

Some might not be satisfied with our Coq assumptions. All uses of additional axioms in our formalization arise from the need to prove *propositional equality* of various Coq objects. A standard way to remedy this situation is to replace sets/types with *setoids* [6], also called *Bishop sets*. The main idea is to equip types with an equivalence relation, which we think of as a customized notion of equality. Then, instead of proving propositional equality of elements, we only prove

that they are related by the relation corresponding to their type. Thus, by giving appropriate equivalence relations for `Morphism`, `Exp_obj`, and `SOC_obj`, we can eliminate our reliance on the axioms mentioned so far.

A downside of this approach is that we would also need to show that functions preserve the respective equivalence relations, resulting in a longer and more complicated formalization. For instance, the `Object` type defined in Section 4.3 would become

```
Record Object : Type :=
  { obj : nat → Type
  ; obj_eq : ∀ n, obj n → obj n → Prop
  ; obj_eq_equiv : ∀ n, Equivalence (obj_eq n)
  ; restr : ∀ n, obj (S n) → obj n
  ; restr_resp : ∀ n x y, obj_eq (S n) x y → obj_eq n (restr n x) (restr n y)
  }.
```

where `Equivalence (obj_eq n)` states that `obj_eq n` is an equivalence relation. The first three fields together express that we have a family of setoids indexed by the natural numbers. The field `restr_resp` witnesses the fact that restriction respects the equivalence relations. We would need to provide proofs of similar statements for most, if not all, of the operations, including the components of morphisms. To keep the formalization simple, we have decided to stick with the axioms.

We could get rid of the dependence on the proof-irrelevance axiom by using an `SProp` version of the equality type in `Morphism`, `Exp_obj`, and `SOC_obj`. Since we do not rely on definitional equalities between these objects, doing so would not improve the ease of formalization. It would, however, require some boilerplate for proving various lemmas for the new equality type not present in the Coq standard library. For these reasons, we stick with the proof-irrelevance axiom.

5 Related work

Sieczkowski et al. created the ModuRes Coq library [29], which provides a general framework for solving recursive domain equations, and can be used to build models of complex programming languages and program logics. The mathematical context for their library is given by ordered families of equivalences (OFEs) [18] and $\mathcal{M}$-categories [12]. They showed that the topos of trees is an $\mathcal{M}$-category, meaning that it provides an appropriate setting for solving recursive domain equations. For this purpose, they only needed a small portion of the structure of the topos of trees. Hence, our formalization of the topos of trees and its logic is more extensive in this regard. However, they also showed the existence of unique fixed points for locally contractive functors, allowing them to model guarded recursive types missing from our formalization. Furthermore, they avoid axioms by using setoids (cf. Section 4.6).

Bergwerf [7] proved a completeness result for propositional logic with the later modality with respect to the simple step-indexed model of Section 3.1. In addition to the rules presented in this report, his completeness theorem depends on an extra axiom, called the comparison rule, which carries a classical flavor. In particular, his calculus does not satisfy the disjunction property, a traditional intuitionistic principle. He also formalized his results in Coq, using a deep embedding for the syntax of the logic.

The use of the topos of trees as denotational semantics for guarded recursion was first suggested by Birkedal et al [10, 11]. They considered not only guarded recursive functions, but also guarded recursive types, for which they generalized the results of Birkedal, Støvring, and Thamsborg [12]. Subsequently, Birkedal and Møgelberg [9] showed that guarded recursive types could be constructed internally by taking fixed points of functions on universes.

Clouston et al. [15] introduced the program logic $Lg\lambda$ for reasoning about terms of the guarded lambda calculus ($g\lambda$), an extension of the simply typed lambda calculus with guarded recursion and coinduction. Our presentation of the internal logic of $\mathcal{S}$ is largely based on $Lg\lambda$. In addition to guarded recursive types, $g\lambda$ also supports coinductive types via a second modality $\blacksquare$, allowing the definition of acausal functions such as `even` from Remark 2.2.10. An alternative approach to extending guarded recursive types to coinductive types is provided by *clock quantifiers* [5, 14, 26].

Bizjak et al. [13] presented guarded dependent type theory (gDTT), a full-fledged dependent type theory with Π-types, Σ-types, extensional identity types, and universes, augmented with the

▶ modality, guarded recursion, and clock quantifiers. As is usual with dependent type theories, their system integrates the logic into the core term calculus via the Curry-Howard correspondence. They also generalized the applicative structure of ▶ to dependent types, which is crucial for both programming and proving. Building on this work, Birkedal et al. [8] combined gDTT with cubical type theory [16], resulting in an intensional type theory with an improved treatment of equality and decidable type checking.

6 Conclusion and future work

We have provided an in-depth exposition of the topos of trees, including how it can be used to model guarded recursive functions, as well as its internal logic. We have argued that the `lift` connective should replace the ▷ modality as a primitive connective, due to its necessity for defining guarded recursive predicates and generalizing earlier axioms involving ▷ and quantifiers. Finally, we have formalized most constructions and rules presented in the report in the Coq proof assistant, discussing technical insights and design choices.

The study of the `lift` connective is far from complete. If `lift` is to replace the ▷ modality, then the original rules involving ▷ must be generalized to use `lift` as well. However, it is not clear how this can be achieved. For instance, the monotonicity rule (▷-MONO) should be replaced by a rule similar to the following:

$$\frac{\Gamma \mid P \vdash Q}{\Gamma \mid \texttt{lift}\, P \vdash \texttt{lift}\, Q}\ \texttt{lift}\text{-MONO}$$

The issue is that P and Q have type ▶`Prop` instead of `Prop`, so the logical consequence $\Gamma \mid P \vdash Q$ is undefined. A possible remedy is to define a separate entailment relation between terms of type ▶`Prop`, along with rules allowing us to move between the two kinds of judgments. This approach is reminiscent of the philosophy behind modal type theory (MTT) [20], where multiple copies of some core type theory are interacting as prescribed by a 'mode' theory. This idea is reinforced by the fact that guarded dependent type theory can be obtained by instantiating MTT.

As alluded to in the introduction, step-indexed logics can be used to construct models of programming languages and program logics with self-referential features, such as Iris [22]. It would be worthwhile to attempt to formalize a model of Iris in the internal logic of the topos of trees. It would be especially interesting to carry out the formalization in a proof assistant which implements guarded recursion natively, such as guarded dependent type theory [13] or guarded cubical type theory [8].

Acknowledgements. I am grateful to Robbert Krebbers for his patience and kindness, for his feedback and helpful suggestions on this report, for the interesting discussions during our meetings, and for sharing his Coq expertise.

References

[1] Amal Jamil Ahmed. *Semantics of Types for Mutable State.* Thesis, Princeton University, November 2004.

[2] Andrew W Appel. Foundational proof-carrying code. In *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science*, pages 247–256. IEEE, 2001.

[3] Andrew W. Appel and David A. McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(5):657–683, 2001.

[4] Andrew W. Appel, Paul-André Melliès, Christopher D. Richards, and Jérôme Vouillon. A very modal model of a modern, major, general type system. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, page 109–122, New York, NY, USA, 2007. Association for Computing Machinery.

[5] Robert Atkey and Conor McBride. Productive coprogramming with guarded recursion. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 197–208. ACM, 2013.

[6] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, March 2003.

[7] Herman Bergwerf. Circular lists in iris * deduction rules of $\triangleright$. Internship report, Radboud University, March 2022.

[8] Lars Birkedal, Aleš Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded cubical type theory. *Journal of Automated Reasoning*, 63(2):211–253, August 2019.

[9] Lars Birkedal and Rasmus Ejlers Møgelberg. Intensional type theory with guarded recursive types qua fixed points on universes. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 213–222. IEEE Computer Society, 2013.

[10] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: Step-indexing in the topos of trees. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 55–64. IEEE, 2011.

[11] Lars Birkedal, Rasmus Ejlers Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, Volume 8, Issue 4, October 2012.

[12] Lars Birkedal, Kristian Støvring, and Jacob Thamsborg. The category-theoretic solution of recursive metric-space equations. *Theoretical Computer Science*, 411(47):4102–4122, 2010.

[13] Aleš Bizjak, Hans Bugge Grathwohl, Ranald Clouston, Rasmus Ejlers Møgelberg, and Lars Birkedal. Guarded dependent type theory with coinductive types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 20–35, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[14] Aleš Bizjak and Rasmus Ejlers Møgelberg. A model of guarded recursion with clock synchronisation. In Dan R. Ghica, editor, *The 31st Conference on the Mathematical Foundations of Programming Semantics, MFPS 2015, Nijmegen, The Netherlands, June 22-25, 2015*, volume 319 of *Electronic Notes in Theoretical Computer Science*, pages 83–101. Elsevier, 2015.

[15] Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. The guarded lambda-calculus: Programming and reasoning with guarded recursion for coinductive types. *Logical Methods in Computer Science*, Volume 12, Issue 3, April 2017.

[16] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[17] Roy L. Crole. *Categories for Types*. Cambridge mathematical textbooks. Cambridge University Press, 1993.

[18] Pietro Di Gianantonio and Marino Miculan. A unifying approach to recursive and co-recursive definitions. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, volume 2646 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 2002.

[19] Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–28, January 2019.

[20] Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. *Logical Methods in Computer Science*, 17(3), 2021.

[21] Bart Jacobs. *Categorical logic and type theory*, volume 141 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, January 1999.

[22] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[23] Joachim Lambek and Philip J. Scott. *Introduction to higher-order categorical logic*. Cambridge University Press, 1986.

[24] Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Universitext. Springer New York, New York, NY, US, May 1992.

[25] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1—-13, 2008.

[26] Rasmus Ejlers Møgelberg. A type theory for productive coprogramming via guarded recursion. In Thomas A. Henzinger and Dale Miller, editors, *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014*, pages 71:1–71:10. ACM, 2014.

[27] Hiroshi Nakano. A modality for recursion. In *Proceedings Fifteenth Annual IEEE Symposium on Logic in Computer Science (Cat. No.99CB36332)*, pages 255–266. IEEE, 2000.

[28] Jan J. M. M. Rutten. Behavioural differential equations: a coinductive calculus of streams, automata, and power series. *Theoretical Computer Science*, 308(1-3):1–53, 2003.

[29] Filip Sieczkowski, Aleš Bizjak, and Lars Birkedal. ModuRes: A Coq library for modular reasoning about concurrent higher-order imperative programming languages. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 375–390, Cham, 2015. Springer International Publishing.

[30] Matthieu Sozeau. *Un environnement pour la programmation avec types dépendants*. PhD thesis, Université Paris Sud - Paris XI, December 2008.

[31] Paul Taylor. *Practical Foundations of Mathematics*, volume 59 of *Cambridge studies in advanced mathematics*. Cambridge University Press, 1999.

[32] The Coq Development Team. The Coq proof assistant, version 8.15.2, January 2022.

[33] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[34] Verified Software Toolchain Team. Verified software toolchain. https://vst.cs.princeton.edu/. Accessed: 2023-07-23.

[35] Martin Wildmoser and Tobias Nipkow. Certifying machine code safety: Shallow versus deep embedding. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics*, volume 3223 of *Lecture Notes in Computer Science*, pages 305–320, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Appendix

$$\text{VAR}\ \frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \text{MOR}\ \frac{f : A \to B \text{ in } \mathcal{S}}{\Gamma \vdash f : A \to B} \qquad \text{UNIT}\ \Gamma \vdash () : \mathbf{1}$$

$$\text{PAIR}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash (t, u) : A \times B} \qquad \text{PROJ1}\ \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_1 t : A} \qquad \text{PROJ2}\ \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \pi_2 t : B}$$

$$\text{ABORT}\ \frac{\Gamma \vdash t : \mathbf{0}}{\Gamma \vdash \mathtt{abort}\, t : A} \qquad \text{INJ1}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{inj}_1\, t : A + B} \qquad \text{INJ2}\ \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathtt{inj}_2\, t : A + B}$$

$$\text{CASE}\ \frac{\Gamma \vdash s : A + B \qquad \Gamma, x : A \vdash t : C \qquad \Gamma, y : B \vdash u : C}{\Gamma \vdash \mathtt{case}\, s\, \mathtt{of}\, x.t; y.u : C}$$

$$\text{LAM}\ \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.t : A \to B} \qquad \text{APP}\ \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}$$

$$\text{NEXT}\ \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{next}\, t : \blacktriangleright A} \qquad \text{AP}\ \frac{\Gamma \vdash t : \blacktriangleright(A \to B) \qquad \Gamma \vdash u : \blacktriangleright A}{\Gamma \vdash t \circledast u : \blacktriangleright B} \qquad \text{REC}\ \frac{\Gamma, x : \blacktriangleright A \vdash t : A}{\Gamma \vdash \mu x : \blacktriangleright A.t : A}$$

$$\text{TRUE}\ \Gamma \vdash \top : \mathtt{Prop} \qquad \text{FALSE}\ \Gamma \vdash \bot : \mathtt{Prop} \qquad \text{EQ}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t =_A u : \mathtt{Prop}} \qquad \text{LATER}\ \frac{\Gamma \vdash P : \mathtt{Prop}}{\Gamma \vdash \triangleright P : \mathtt{Prop}}$$

$$\text{CONJ}\ \frac{\Gamma \vdash P : \mathtt{Prop} \qquad \Gamma \vdash Q : \mathtt{Prop}}{\Gamma \vdash P \wedge Q : \mathtt{Prop}} \qquad \text{DISJ}\ \frac{\Gamma \vdash P : \mathtt{Prop} \qquad \Gamma \vdash Q : \mathtt{Prop}}{\Gamma \vdash P \vee Q : \mathtt{Prop}}$$

$$\text{IMPL}\ \frac{\Gamma \vdash P : \mathtt{Prop} \qquad \Gamma \vdash Q : \mathtt{Prop}}{\Gamma \vdash P \supset Q : \mathtt{Prop}} \qquad \text{FORALL}\ \frac{\Gamma, x : A \vdash P : \mathtt{Prop}}{\Gamma \vdash \forall x : A.\, P : \mathtt{Prop}} \qquad \text{EXISTS}\ \frac{\Gamma, x : A \vdash P : \mathtt{Prop}}{\Gamma \vdash \exists x : A.\, P : \mathtt{Prop}}$$

Figure 4: Typing rules for $\mathcal{L}$

$$\text{REFL}\ \ \Gamma \mid P \vdash P$$

$$\text{TRANS}\ \ \frac{\Gamma \mid P \vdash Q \qquad \Gamma \mid Q \vdash R}{\Gamma \mid P \vdash R}$$

$$\text{WK}\ \ \frac{\Gamma \mid P \vdash Q}{\Gamma, x : A \mid P \vdash Q}$$

$$\text{SUBST}\ \ \frac{\Gamma, x : A \mid P \vdash Q \qquad \Gamma \vdash t : A}{\Gamma \mid P[x := t] \vdash Q[x := t]}$$

$$\top\text{-INTRO}\ \ \Gamma \mid P \vdash \top$$

$$\bot\text{-ELIM}\ \ \Gamma \mid \bot \vdash P$$

$$\wedge\text{-INTRO}\ \ \frac{\Gamma \mid R \vdash P \qquad \Gamma \mid R \vdash Q}{\Gamma \mid R \vdash P \wedge Q}$$

$$\wedge\text{-ELIM-L}\ \ \Gamma \mid P \wedge Q \vdash P$$

$$\wedge\text{-ELIM-R}\ \ \Gamma \mid P \wedge Q \vdash Q$$

$$\vee\text{-INTRO-L}\ \ \Gamma \mid P \vdash P \vee Q$$

$$\vee\text{-INTRO-R}\ \ \Gamma \mid Q \vdash P \vee Q$$

$$\vee\text{-ELIM}\ \ \frac{\Gamma \mid P \vdash R \qquad \Gamma \mid Q \vdash R}{\Gamma \mid P \vee Q \vdash R}$$

$$\supset\text{-INTRO}\ \ \frac{\Gamma \mid R \wedge P \vdash Q}{\Gamma \mid R \vdash P \supset Q}$$

$$\supset\text{-ELIM}\ \ \Gamma \mid (P \supset Q) \wedge P \vdash Q$$

$$\forall\text{-INTRO}\ \ \frac{\Gamma, x : A \mid R \vdash P}{\Gamma \mid R \vdash \forall x : A.\, P}$$

$$\forall\text{-ELIM}\ \ \frac{\Gamma \vdash t : A}{\Gamma \mid \forall x : A.\, P \vdash P[x := t]}$$

$$\exists\text{-INTRO}\ \ \frac{\Gamma \vdash t : A}{\Gamma \mid P[x := t] \vdash \exists x : A.\, P}$$

$$\exists\text{-ELIM}\ \ \frac{\Gamma, x : A \mid P \vdash Q}{\Gamma \mid \exists x : A.\, P \vdash Q}$$

$$\triangleright\text{-INTRO}\ \ \Gamma \mid P \vdash \triangleright P$$

$$\triangleright\text{-MONO}\ \ \frac{\Gamma \mid P \vdash Q}{\Gamma \mid \triangleright P \vdash \triangleright Q}$$

$$\text{LÖB}\ \ \frac{\Gamma \mid \triangleright P \vdash P}{\Gamma \vdash P}$$

$$\wedge\text{-}\triangleright\ \ \Gamma \mid \triangleright P \wedge \triangleright Q \vdash \triangleright(P \wedge Q)$$

$$\triangleright\text{-}\vee\ \ \Gamma \mid \triangleright(P \vee Q) \vdash \triangleright P \vee \triangleright Q$$

$$\supset\text{-}\triangleright\ \ \Gamma \mid \triangleright P \supset \triangleright Q \vdash \triangleright(P \supset Q)$$

$$\triangleright\text{-EQ}\ \ \Gamma \mid \triangleright(t =_A u) \dashv\vdash \texttt{next}\, t =_{\blacktriangleright A} \texttt{next}\, u$$

Figure 5: Structural and logical rules for $\mathcal{L}$

EQ-REFL
$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t =_A t}$$

EQ-TRANS
$$\frac{\Gamma \vdash t =_A u \qquad \Gamma \vdash u =_A v}{\Gamma \vdash t =_A v}$$

EQ-SYM
$$\frac{\Gamma \vdash t =_A u}{\Gamma \vdash u =_A t}$$

EQ-SUBST
$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u =_A v}{\Gamma \vdash t[x := u] =_B t[x := v]}$$

EQ-PROP
$$\frac{\Gamma \vdash P =_{\mathtt{Prop}} Q}{\Gamma \mid P \vdash Q}$$

PROPEXT
$$\frac{\Gamma \mid P \vdash Q \qquad \Gamma \mid Q \vdash P}{\Gamma \vdash P =_{\mathtt{Prop}} Q}$$

$\mathbf{1}$-η
$$\frac{\Gamma \vdash t : \mathbf{1}}{\Gamma \vdash t =_{\mathbf{1}} ()}$$

$\times$-β-1
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \pi_1(t, u) =_A t}$$

$\times$-β-2
$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \pi_2(t, u) =_B u}$$

$\times$-η
$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash (\pi_1 t, \pi_2 t) =_{A \times B} t}$$

+-β-1
$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A \vdash u : C \qquad \Gamma, y : B \vdash v : C}{\Gamma \vdash \mathtt{case}\ \mathtt{inj}_1\, t\ \mathtt{of}\ x.u; y.v =_C u[x := t]}$$

+-β-2
$$\frac{\Gamma \vdash t : B \qquad \Gamma, x : A \vdash u : C \qquad \Gamma, y : B \vdash v : C}{\Gamma \vdash \mathtt{case}\ \mathtt{inj}_2\, t\ \mathtt{of}\ x.u; y.v =_C v[y := t]}$$

$\to$-β
$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash u : A}{\Gamma \vdash (\lambda x : A.\, t)\, u =_B t[x := u]}$$

$\to$-η
$$\frac{\Gamma \vdash t : A \to B \qquad (x : A) \notin \Gamma}{\Gamma \vdash \lambda x : A.\, t\, x =_{A \to B} t}$$

$\circledast$-NEXT
$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathtt{next}\, t \circledast \mathtt{next}\, u =_{\blacktriangleright B} \mathtt{next}\, (t\, u)}$$

$\circledast$-COMP
$$\frac{\Gamma \vdash f : \blacktriangleright(B \to C) \qquad \Gamma \vdash g : \blacktriangleright(A \to B) \qquad \Gamma \vdash t : \blacktriangleright A}{\Gamma \vdash f \circledast (g \circledast t) =_{\blacktriangleright C} \mathtt{next}\ \mathtt{comp} \circledast f \circledast g \circledast t}$$

FIX-β
$$\frac{\Gamma, x : \blacktriangleright A \vdash t : A}{\Gamma \vdash \mu x : \blacktriangleright A.\, t =_A t[x := \mathtt{next}\, (\mu x : \blacktriangleright A.\, t)]}$$

FIX-η
$$\frac{\Gamma, x : \blacktriangleright A \vdash t : A \qquad \Gamma \vdash z : A \qquad \Gamma \vdash z = t[x := \mathtt{next}\, z]}{\Gamma \vdash z =_A \mu x : \blacktriangleright A.\, t}$$

Figure 6: Equality rules in $\mathcal{L}$. $\mathtt{comp} : (B \to C) \to (A \to B) \to A \to C$ is function composition. Congruence cases are omitted.